

PYTHON

PROGRAMMING

BASIC GUIDE FOR B E G I N N E R



E L E N A M E R R Y

PYTHON BASIC PROGRAMMING GUIDE FOR BEGINNERS

BY ELENA MERRY

Python Basic Programming Guide

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher Elena Merry.

About Author

Elena Merry (born 1971) Miami is a high school science teacher and math teacher living in Alaska where he also teaches introductory programming classes. He has been writing programs since he was five years old and is the author of the best-selling Python Intro

1.1 Installing Python

CONTENT

1 INTRODUCTION BASICS OF PYTHON 2 PYTHON BASIC FUNCTIONS 3 STORING AND MODIFYING INFORMATION 4 WORKING ON TEXT DATA 5 DEEP LEARNING ADVANCED API 6 FIXED REGRESSIONS 1 INTRODUCTION BASICS OF PYTHON

1.1 Installing Python

Go to www.python.org and download the latest version of Python (version 3.5 as of this writing). It should be painless to install. If you have a Mac or Linux, you may already have Python on your computer, though it may be an older version. If it is version 2.7 or earlier, then you should install the latest version, as many of the programs in this book will not work correctly on older versions.

1.2 IDLE

IDLE is a simple integrated development environment (IDE) that comes with Python. It's a program that allows you to type in your programs and run them. There are other IDEs for Python, but for now I would suggest sticking with IDLE as it is simple to use. You can find IDLE in the Python 3.4 folder on your computer.

When you first start IDLE, it starts up in the shell, which is an interactive window where you can type in Python code and see the output in the same window. I often use the shell in place of my calculator or to try out small pieces of code. But most of the time you will want to open up a new window and type the program in there.

Note At least on Windows, if you click on a Python file on your desktop, your system will run the program, but not show the code, which is probably not what you want. Instead, if you right-click on the file, there should be an

option called Edit with Idle. To edit an existing Python file,

3

either do that or start up IDLE and open the file through the File menu.

Keyboard shortcuts speed up your work. The following keystrokes work in IDLE and can really

Keystroke Result

CTRL+C Copy selected text

CTRL+X Cut selected text

CTRL+V Paste

CTRL+Z Undo the last keystroke or group of keystrokes

CTRL+SHIFT+Z Redo the last keystroke or group of keystrokes

F5

1.3 A first program Run module

Start IDLE and open up a new window (choose New Window under the File Menu). Type in the following program.

```
temp = eval(input('Enter a temperature in Celsius: ')) print('In Fahrenheit,  
that is', 9/5*temp+32)
```

Then, under the Run menu, choose Run Module (or press F5). IDLE will ask you to save the file, and you should do so. Be sure to append .py to the filename as IDLE will not automatically append it. This will tell IDLE to use colors to make your program easier to read.

Once you've saved the program, it will run in the shell window. The program will ask you for a temperature. Type in 20 and press enter. The program's output looks something like this:

```
Enter a temperature in Celsius: 20
```

```
In Fahrenheit, that is 68.0
```

Let's examine how the program does what it does. The first line asks the user to enter a temperature. The **input** function's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a *string* and it will appear to the program's user exactly as it appears in the code itself. The **eval** function is something we use here, but it won't be clear exactly why until later. So for

now, just remember that we use it when we're getting numerical input.

We need to give a name to the value that the user enters so that the program can remember it and use it in the second line. The name we use is temp and we use the equals sign to assign the user's value to temp.

The second line uses the **print** function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here. The second

1.4. TYPING THINGS IN

argument to the **print** function is the calculation. Python will do the calculation and print out the numerical result.

This program may seem too short and simple to be of much use, but there are many websites that have little utilities that do similar conversions, and their code is not much more complicated than the code here.

A second program Here is a program that computes the average of two numbers that the user enters:

```
num1 = eval(input('Enter the first number: ')) num2 = eval(input('Enter the  
second number: '))  
print('The average of the numbers you entered is', (num1+num2)/2)
```

For this program we need to get two numbers from the user. There are ways to do that in one line, but for now we'll keep things simple. We get the numbers one at a time and give each number its own name. The only other thing to note is the parentheses in the average calculation. This is because of the order of operations. All multiplications and divisions are performed before any additions and subtractions, so we have to use parentheses to get Python to do the addition first.

1.4 Typing things in

Case Case matters. To Python, print, Print, and PRINT are all different things. For now, stick with lowercase as most Python statements are in lowercase. **Spaces** Spaces matter at the beginning of lines, but not elsewhere. For example, the code below will not work.

```
temp = eval(input('Enter a temperature in Celsius: '))
```

```
print('In Fahrenheit, that is', 9/5*temp+32)
```

Python uses indentation of lines for things we'll learn about soon. On the other hand, spaces in most other places don't matter. For instance, the following lines have the same effect:

```
print('Hello world!') print ('Hello world!') print( 'Hello world!' )
```

Basically, computers will only do what you tell them, and they often take things very literally. Python itself totally relies on things like the placement of commas and parentheses so it knows what's what. It is not very good at figuring out what you mean, so you have to be precise. It will be very frustrating at first, trying to get all of the parentheses and commas in the right places, but after a while it will become more natural. Still, even after you've programmed for a long time, you will still miss something. Fortunately, the Python interpreter is pretty good about helping you find your mistakes.

1.5 Getting input

The **input** function is a simple way for your program to get information from people using your program. Here is an example:

```
name = input('Enter your name: ') print('Hello, ', name)
```

The basic structure is *variable name* = **input**(*message to user*)

The above works for getting text from the user. To get numbers from the user to use in calculations, we need to do something extra. Here is an example:

```
num = eval(input('Enter a number: ')) print('Your number squared:',  
num*num)
```

The **eval** function converts the text entered by the user into a number. One nice feature of this is you can enter expressions, like $3*12+5$, and **eval** will compute them for you.

Note If you run your program and nothing seems to be happening, try pressing enter. There is a bit of a glitch in IDLE that occasionally happens with **input** statements.

1.6 Printing

Here is a simple example:

```
print('Hi there')
```

The **print** function requires parenthesis around its arguments. In the program above, its only argument is the string 'Hi there'. Anything inside quotes will (with a few exceptions) be printed exactly as it appears. In the following, the first statement will output 3+4, while the second will output 7.

```
print('3+4') print(3+4)
```

To print several things at once, separate them by commas. Python will automatically insert spaces between them. Below is an example and the output it produces.

```
print ('The value of 3+4 is', 3+4) print('A', 1, 'XYZ', 2)
```

The value of 3+4 is 7

A 1 XYZ 2

1.7. VARIABLES

Optional arguments

There are two optional arguments to the **print** function. They are not overly important at this stage of the game, so you can safely skip over this section, but they are useful for making your output look nice.

sep Python will insert a space between each of the arguments of the print function. There is an optional argument called sep, short for separator, that you can use to change that space to something else. For example, using sep=':' would separate the arguments by a colon and sep='##' would separate the arguments by two pound signs.

One particularly useful possibility is to have nothing inside the quotes, as in sep="". This says to put no separation between the arguments. Here is an example where sep is useful for getting the output to look nice:

```
print ('The value of 3+4 is', 3+4, '.') print ('The value of 3+4 is ', 3+4, '.',  
sep="")
```

The value of 3+4 is 7 .

The value of 3+4 is 7.

end The print function will automatically advance to the next line. For instance, the following will print on two lines:

```
print('On the first line') print('On the second line')
```

On the first line

On the second line

There is an optional argument called end that you can use to keep the print function from advancing to the next line. Here is an example:

```
print('On the first line', end="") print('On the second line')
```

On the first lineOn the second line

Of course, this could be accomplished better with a single print, but we will see later that there are interesting uses for the end argument.

1.7 Variables

Looking back at our first program, we see the use of a variable called temp:

```
temp = eval(input('Enter a temperature in Celsius: ')) print('In Fahrenheit,  
that is', 9/5*temp+32)
```

One of the major purposes of a variable is to remember a value from one part of a program so that it can be used in another part of the program. In the case above, the variable temp stores the value that the user enters so that we can do a calculation with it in the next line.

In the example below, we perform a calculation and need to use the result of the calculation in several places in the program. If we save the result of the calculation in a variable, then we only need to do the calculation once. This also helps to make the program more readable.

```
temp = eval(input('Enter a temperature in Celsius: ')) f_temp = 9/5*temp+32  
print('In Fahrenheit, that is', f_temp) if f_temp > 212: print('That  
temperature is above the boiling point.')
```

```
if f_temp < 32:
```

```
print('That temperature is below the freezing point.')
```

We haven't discussed if statements yet, but they do exactly what you think they do.

A second example Here is another example with variables. Before reading on, try to figure out what the values of x and y will be after the code is executed. x=3 y=4 z=x+y z=z+1 x=y y=5

After these four lines of code are executed, x is 4, y is 5 and z is 8. One way to understand something like this is to take it one line at a time. This is an especially useful technique for trying to understand more complicated chunks of code. Here is a description of what happens in the code above:

x starts with the value 3 and y starts with the value 4.
In line 3, a variable z is created to equal x+y, which is 7.

Then the value of z is changed to equal one more than it currently equals, changing it from 7 to 8.

Next, x is changed to the current value of y, which is 4.
Finally, y is changed to 5. Note that this does not affect x.
So at the end, x is 4, y is 5, and z is 8.

1.8. EXERCISES

Variable names

There are just a couple of rules to follow when naming your variables. Variable names can contain letters, numbers, and the underscore. Variable names *cannot* contain spaces.

Variable names *cannot* start with a number.

Case matters—for instance, temp and Temp are different.

It helps make your program more understandable if you choose names that are descriptive, but not so long that they clutter up your program.

1.8 Exercises

Print a box like the one below. *****

Print a box like the one below. *****

*

*

Print a triangle like the one below. *

**

Write a program that computes and prints the result of



. It is roughly .1017.

Ask the user to enter a number. Print out the square of the number, but use the sep optional argument to print it out in a full sentence that ends in a period. Sample output is shown below.

Enter a number: 5 The square of 5 is 25.

Ask the user to enter a number x . Use the sep optional argument to print out x , $2x$, $3x$, $4x$, and $5x$, each separated by three dashes, like below.

Enter a number: 7

7---14---21---28---35

Write a program that asks the user for a weight in kilograms and converts it to pounds. There are 2.2 pounds in a kilogram.

Write a program that asks the user to enter three numbers (use three separate input statements). Create variables called total and average that hold the sum and average of the three numbers and print out the values of total and average.

A lot of cell phones have tip calculators. Write one. Ask the user for the price of the meal and the percent tip they want to leave. Then print both the tip amount and the total bill with the tip included.

For loops

Probably the most powerful thing about computers is that they can repeat things over and over very quickly. There are several ways to repeat things in Python, the most common of which is the for loop.

2.1 Examples

Example 1 The following program will print Hello ten times:

```
for i in range(10):  
    print('Hello')
```

The structure of a for loop is as follows: **for** *variable name* **in range**(*number of times to repeat*): *statements to be repeated*

The syntax is important here. The word **for** must be in lowercase, the first line must end with a colon, and the statements to be repeated *must* be indented. Indentation is used to tell Python which statements will be repeated.

Example 2 The program below asks the user for a number and prints its square, then asks for another number and prints its square, etc. It does this three times and then prints that the loop is done.

```
for i in range(3):  
    num = eval(input('Enter a number: '))  
    print ('The square of your number is',  
           num*num)
```

```
print ('The loop is now done.')
```

11

Enter a number: 3

The square of your number is 9

Enter a number: 5

The square of your number is 25

Enter a number: 23

The square of your number is 529 The loop is now done.

Since the second and third lines are indented, Python knows that these are the statements to be repeated. The fourth line is not indented, so it is not part of the loop and only gets executed once, after the loop has completed.

Looking at the above example, we see where the term *for loop* comes from: we can picture the execution of the code as starting at the **for** statement, proceeding to the second and third lines, then looping back up to the **for** statement.

Example 3 The program below will print A, then B, then it will alternate C's and D's five times and then finish with the letter E once.

```
print ('A')  
print('B')  
for i in range(5):  
    print('C')  
    print('D')
```

```
print('E')
```

The first two print statements get executed once, printing an A followed by a B. Next, the C's and D's alternate five times. Note that we don't get five C's followed by five D's. The way the loop works is we print a C, then a D, then loop back to the start of the loop and print a C and another D, etc. Once the program is done looping with the C's and D's, it prints one E.

Example 4 If we wanted the above program to print five C's followed by five D's, instead of alternating C's and D's, we could do the following:

```
print('A') print('B') for i in range(5):  
print('C')  
for i in range(5):  
print('D')  
print('E')
```

2.2. THE LOOP VARIABLE

2.2 The loop variable

There is one part of a for loop that is a little tricky, and that is the loop variable. In the example below, the loop variable is the variable *i*. The output of this program will be the numbers 0, 1, ..., 99, each printed on its own line.

```
for i in range(100):  
print(i)
```

When the loop first starts, Python sets the variable *i* to 0. Each time we loop back up, Python increases the value of *i* by 1. The program loops 100 times, each time increasing the value of *i* by 1, until we have looped 100 times. At this point the value of *i* is 99.

You may be wondering why *i* starts with 0 instead of 1. Well, there doesn't seem to be any really good reason why other than that starting at 0 was useful in the early days of computing and it has stuck with us. In fact most things in computer programming start at 0 instead of 1. This does take some getting used to.

Since the loop variable, `i`, gets increased by 1 each time through the loop, it can be used to keep track of where we are in the looping process. Consider the example below:

```
for i in range(3):
    print(i+1, '-- Hello')
-- Hello
-- Hello
-- Hello
```

Names There's nothing too special about the name `i` for our variable. The programs below will have the exact same result.

```
for i in range(100):
    for wacky_name in range(100):
        print(i)
        print(wacky_name)
```

It's a convention in programming to use the letters `i`, `j`, and `k` for loop variables, unless there's a good reason to give the variable a more descriptive name.

2.3 The range function

The value we put in the `range` function determines how many times we will loop. The way `range` works is it produces a list of numbers from zero to the value minus one. For instance, `range(5)` produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement `range(1,5)` will produce the list 1, 2, 3, 4. This brings up one quirk of the `range` function—it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do `range(1,6)`.

Another thing we can do is to get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement `range(1,10,2)` will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1. For instance, `range(5,1,-1)` will produce the values 5, 4, 3, 2, in that order. (Note that the `range` function stops one short of the ending value 1). Here are a few more examples:

Statement Values generated

range(10) 0,1,2,3,4,5,6,7,8,9

range(1,10) 1,2,3,4,5,6,7,8,9

range(3,7) 3,4,5,6

range(2,15,3) 2,5,8,11,14

range (9,2,-1) 9,8,7,6,5,4,3

Here is an example program that counts down from 5 and then prints a message. **for** i **in** **range**(5,0,-1):

print(i, end=' ')

print('Blast off!!!')

5 4 3 2 1 Blast off!!!

The end=" just keeps everything on the same line.

2.4 A Trickier Example

Let's look at a problem where we will make use of the loop variable. The program below prints a rectangle of stars that is 4 rows tall and 6 rows wide.

for i **in** **range**(4):

print(''*6)

The rectangle produced by this code is shown below on the left. The code `''*6` is something we'll cover in Section 6.2; it just repeats the asterisk character six times.

```
***** *
```

```
***** **
```

```
***** ***
```

```
***** ****
```

2.5. EXERCISES

Suppose we want to make a triangle instead. We can accomplish this with a very small change to the rectangle program. Looking at the program, we can see that the for loop will repeat the **print** statement four times, making the shape four rows tall. It's the 6 that will need to change.

The key is to change the 6 to i+1. Each time through the loop the program

will now print $i+1$ stars instead of 6 stars. The loop counter variable i runs through the values 0, 1, 2, and 3. Using it allows us to vary the number of stars. Here is triangle program:

```
for i in range(4): print('*'*(i+1))
```

2.5 Exercises

Write a program that prints your name 100 times.

Write a program to fill the screen horizontally and vertically with your name. [Hint: add the option `end=""` into the `print` function to fill the screen horizontally.] Write a program that outputs 100 lines, numbered 1 to 100, each with your name on it. The output should look like the output below.

```
Your name
Your name
Your name
Your name ...
100 Your name
```

Write a program that prints out a list of the integers from 1 to 20 and their squares. The output should look like this:

```
--- 1
--- 4
--- 9 ...
```

```
20 --- 400
```

Write a program that uses a for loop to print the numbers 8, 11, 14, 17, 20, ..., 83, 86, 89.

Write a program that uses a for loop to print the numbers 100, 98, 96, ..., 4, 2.

Write a program that uses exactly four for loops to print the sequence of letters below.

```
AAAAAAAAAABBBBBBBBCDCDCDCDEFFFFFFFG
```

Write a program that asks the user for their name and how many times to print it. The program should print out the user's name the specified number of times.

The Fibonacci numbers are the sequence below, where the first two numbers are 1, and each number thereafter is the sum of the two preceding numbers. Write a program that asks the user how many Fibonacci numbers to print and then prints that many.

1,1,2,3,5,8,13,21,34,55,89...

Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be. [Hint: `print('*'*10)` prints ten asterisks.]

```
*****
*****
*****
*****
```

Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be.

```
*****
*
*
*****
```

Use a for loop to print a triangle like the one below. Allow the user to specify how high the triangle should be.

```
*
**
***
****
```

Use a for loop to print an upside down triangle like the one below. Allow the user to specify how high the triangle should be.

```
****
***
**
*
```


Use for loops to print a diamond like the one below. Allow the user to specify how high the diamond should be.

```
*
***
*****
*****
*****
***
*
```

2.5. EXERCISES

Write a program that prints a giant letter A like the one below. Allow the user to specify how large the letter should be.

Numbers

This chapter focuses on numbers and simple mathematics in Python. 3.1
Integers and Decimal Numbers

Because of the way computer chips are designed, integers and decimal numbers are represented differently on computers. Decimal numbers are represented by what are called floating point numbers. The important thing to remember about them is you typically only get about 15 or so digits of precision. It would be nice if there were no limit to the precision, but calculations run a lot more quickly if you cut off the numbers at some point.

On the other hand, integers in Python have no restrictions. They can be arbitrarily large.

For decimal numbers, the last digit is sometimes slightly off due to the fact that computers work in binary (base 2) whereas our human number system is base 10. As an example, mathematically, we know that the decimal expansion of $7/3$ is $2.333\cdots$, with the threes repeating forever. But when we type $7/3$ into the Python shell, we get 2.3333333333333335 . This is called *roundoff error*. For most practical purposes this is not too big of a deal, but it actually can cause problems for some mathematical and scientific calculations. If you really need more precision, there are ways. See [Section 22.5](#).

3.2 Math Operators

Here is a list of the common operators in Python:

19

Operator Description

+ addition

- subtraction

* multiplication

/ division

** exponentiation

// integer division

% modulo (remainder) **Exponentiation** Python uses ** for exponentiation. The caret, ^, is used for something else.

Integer division The integer division operator, //, requires some explanation. Basically, for positive numbers it behaves like ordinary division except that it throws away the decimal part of the result. For instance, while $8/5$ is 1.6, we have $8//5$ equal to 1. We will see uses for this operator later. Note that in many other programming languages and in older versions of Python, the usual division operator / actually does integer division on integers.

Modulo The modulo operator, %, returns the remainder from a division. For instance, the result of $18\%7$ is 4 because 4 is the remainder when 18 is divided by 7. This operation is surprisingly useful. For instance, a number is divisible by n precisely when it leaves a remainder of 0 when divided by n . Thus to check if a number, n , is even, see if $n\%2$ is equal to 0. To check if n is divisible by 3, see if $n\%3$ is 0.

One use of this is if you want to schedule something in a loop to happen only every other time through the loop, you could check to see if the loop variable modulo 2 is equal to 0, and if it is, then do that something.

The modulo operator shows up surprisingly often in formulas. If you need to “wrap around” and come back to the start, the modulo is useful. For example, think of a clock. If you go six hours past 8 o’clock, the result is 2 o’clock. Mathematically, this can be accomplished by doing a modulo by 12. That is, $(8+6)\%12$ is equal to 2.

As another example, take a game with players 1 through 5. Say you have a variable `player` that keeps track of the current player. After player 5 goes, it's player 1's turn again. The modulo operator can be used to take care of this:

```
player = player%5+1
```

When `player` is 5, `player%5` will be 0 and expression will set `player` to 1.

3.3. ORDER OF OPERATIONS

3.3 Order of operations

Exponentiation gets done first, followed by multiplication and division (including `//` and `%`), and addition and subtraction come last. The classic math class mnemonic, PEMDAS (Please Excuse My Dear Aunt Sally), might be helpful.

This comes into play in calculating an average. Say you have three variables `x`, `y`, and `z`, and you want to calculate the average of their values. To expression `x+y+z/3` would not work. Because division comes before addition, you would actually be calculating $x + y + \frac{z}{3}$ instead of $\frac{x+y+z}{3}$. This is easily fixed by using parentheses: `(x+y+z)/3`.

In general, if you're not sure about something, adding parentheses might help and usually doesn't do any harm.

3.4 Random numbers

To make an interesting computer game, it's good to introduce some randomness into it. Python comes with a module, called `random`, that allows us to use random numbers in our programs.

Before we get to random numbers, we should first explain what a *module* is. The core part of the Python language consists of things like for loops, if statements, math operators, and some functions, like **print** and **input**. Everything else is contained in modules, and if we want to use something from a module we have to first *import* it—that is, tell Python that we want to use it.

At this point, there is only one function, called `randint`, that we will need from the `random` module. To load this function, we use the following

statement: **from** random **import** randint

Using randint is simple: randint(a,b) will return a random integer between a and b including both a and b. (Note that randint includes the right endpoint b unlike the **range** function). Here is a short example:

```
from random import randint x = randint(1,10)
print('A random number between 1 and 10: ', x)
```

A random number between 1 and 10: 7

The random number will be different every time we run the program. 3.5

Math functions

The math module Python has a module called math that contains familiar math functions, including sin, cos, tan, exp, log, log10, factorial, sqrt, floor, and ceil. There are also the inverse trig functions, hyperbolic functions, and the constants pi and e. Here is a short example:

```
from math import sin, pi print('Pi is roughly', pi) print('sin(0) =', sin(0))
Pi is roughly 3.14159265359 sin(0) = 0.0
```

Built-in math functions There are two built in math functions, **abs** (absolute value) and **round** that are available without importing the math module. Here are some examples:

```
print (abs(-4.3)) print(round(3.336, 2)) print(round(345.2, -1))
4.3
3.34
350.0
```

The **round** function takes two arguments: the first is the number to be rounded and the second is the number of decimal places to round to. The second argument can be negative.

3.6 Getting help from Python

There is documentation built into Python. To get help on the math module, for example, go to the Python shell and type the following two lines:

```
>>> import math
```

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

This gives a list of all the functions and variables in the math module. You can ignore all of the ones that start with underscores. To get help on a specific function, say the floor function, you can type `help(math.floor)`. Typing `help(math)` will give you help for everything in the math module.

3.7 Using the Shell as a Calculator

The Python shell can be used as a very handy and powerful calculator. Here is an example session:

3.8. EXERCISES

```
>>> 23**2
529
>>> s = 0
>>> for n in range(1,10001):
s = s + 1/n**2
>>> s
1.6448340718480652
>>> from math import *
>>> factorial(10)
3628800
```

The second example here sums the numbers $1 + 1/4 + 1/9 + \dots + 1/10000^2$. The result is stored in the variable `s`. To inspect the value of that variable, just type its name and press enter. Inspecting variables is useful for debugging your programs. If a program is not working properly, you can type your variable names into the shell after the program has finished to see what their values are.

The statement `from math import *` imports every function from the math module, which can make the shell a lot like a scientific calculator.

Note Under the Shell menu, select Restart shell if you want to clear the

values of all the variables.

3.8 Exercises

Write a program that generates and prints 50 random integers, each between 3 and 6.

Write a program that generates a random number, x , between 1 and 50, a random number y between 2 and 5, and computes x^y .

Write a program that generates a random number between 1 and 10 and prints your name that many times.

Write a program that generates a random decimal number between 1 and 10 with two decimal places of accuracy. Examples are 1.23, 3.45, 9.80, and 5.00.

Write a program that generates 50 random numbers such that the first number is between 1 and 2, the second is between 1 and 3, the third is between 1 and 4, ..., and the last is between 1 and 51.

Write a program that asks the user to enter two numbers, x and y , and computes $|xx^{-}+yy|$.

Write a program that asks the user to enter an angle between -180° and 180° . Using an expression with the modulo operator, convert the angle to its equivalent between 0° and 360° .

Write a program that asks the user for a number of seconds and prints out how many minutes and seconds that is. For instance, 200 seconds is 3 minutes and 20 seconds. [Hint: Use the `//` operator to get minutes and the `%` operator to get seconds.]

Write a program that asks the user for an hour between 1 and 12 and for how many hours in the future they want to go. Print out what the hour will be that many hours into the future. An example is shown below.

Enter hour: 8

How many hours ahead? 5

New hour: 1 o'clock

(a) One way to find out the last digit of a number is to mod the number by 10.

Write a program that asks the user to enter a power. Then find the last digit of 2 raised to that power.

One way to find out the last two digits of a number is to mod the number by 100. Write a program that asks the user to enter a power. Then find the last two digits of 2 raised to that power.

Write a program that asks the user to enter a power and how many digits they want. Find the last that many digits of 2 raised to the power the user entered.

Write a program that asks the user to enter a weight in kilograms. The program should convert it to pounds, printing the answer rounded to the nearest tenth of a pound.

Write a program that asks the user for a number and prints out the factorial of that number.

Write a program that asks the user for a number and then prints out the sine, cosine, and tangent of that number.

Write a program that asks the user to enter an angle in degrees and prints out the sine of that angle.

Write a program that prints out the sine and cosine of the angles ranging from 0 to 345° in 15° increments. Each result should be rounded to 4 decimal places. Sample output is shown below:

```
0 --- 0.0 1.0
15 --- 0.2588 0.9659
30 --- 0.5 0.866 ...
345 --- -0.2588 0.9659
```

Below is described how to find the date of Easter in any year. Despite its intimidating appearance, this is not a hard problem. Note that b^xc is the *floor* function, which for positive numbers just drops the decimal part of the number. For instance $b3.14c = 3$. The floor function is part of the math module.

$C = \text{century (1900's)} \rightarrow C = 19$)

3.8. EXERCISES

Y = year (all four digits) $m = (15 + C - b^C_4 c - b^{8C}_{25+13c}) \bmod 30$ $n = (4 + C - b^C_4 c) \bmod 7$ $a = Y \bmod 4$ $b = Y \bmod 7$ $c = Y \bmod 19$ $d = (19c + m) \bmod 30$ $e = (2a + 4b + 6d + n) \bmod 7$

Easter is either March $(22 + d + e)$ or April $(d + e - 9)$. There is an exception if $d = 29$ and $e = 6$.

In this case, Easter falls one week earlier on April 19. There is another exception if $d = 28$, $e = 6$, and $m = 2, 5, 10, 13, 16, 21, 24$, or 39. In this case, Easter falls one week earlier on April 18. Write a program that asks the user to enter a year and prints out the date of Easter in that year. (See Tattersall, *Elementary Number Theory in Nine Chapters*, 2nd ed., page 167)

A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Ask the user to enter a year, and, using the `//` operator, determine how many leap years there have been between 1600 and that year.

Write a program that given an amount of change less than \$1.00 will print out exactly how many quarters, dimes, nickels, and pennies will be needed to efficiently make that change. [Hint: the `//` operator may be useful.]

Write a program that draws “modular rectangles” like the ones below. The user specifies the width and height of the rectangle, and the entries start at 0 and increase typewriter fashion from left to right and top to bottom, but are all done mod 10. Below are examples of a 3×5 rectangle and a 4×8 .

```

0 1 2 3 4
5 6 7 8 9
0 1 2 3 4
0 1 2 3 4 5 6 7
8 9 0 1 2 3 4 5
6 7 8 9 0 1 2 3
4 5 6 7 8 9 0 1
If statements

```

Quite often in programs we only want to do something provided something else is true. Python’s **if** statement is what we need.

4.1 A Simple Example

Let's try a guess-a-number program. The computer picks a random number, the player tries to guess, and the program tells them if they are correct. To see if the player's guess is correct, we need something new, called an *if statement*.

```
from random import randint
num = randint(1,10) guess = eval(input('Enter your guess: ')) if guess==num:
print('You got it!')
```

The syntax of the if statement is a lot like the **for** statement in that there is a colon at the end of the if condition and the following line or lines are indented. The lines that are indented will be executed only if the condition is true. Once the indentation is done with, the if block is concluded.

The guess-a-number game works, but it is pretty simple. If the player guesses wrong, nothing happens. We can add to the if statement as follows:

```
if guess==num:
print('You got it!')
else:
print('Sorry. The number is ', num)
```

We have added an **else** statement, which is like an “otherwise.”

27

4.2 Conditional operators

The comparison operators are ==, >, <, >=, <=, and !=. That last one is for *not equals*. Here are a few examples:

Expression Description

if x>3: if x is greater than 3

if x>=3: if x is greater than or equal to 3

if x==3: if x is 3

if x!=3: if x is not 3

There are three additional operators used to construct more complicated conditions: **and**, **or**, and **not**. Here are some examples:

```
if grade>=80 and grade<90: print('Your grade is a B.')
if score>1000 or time>20:
    print('Game over.')
if not (score>1000 or time>20): print('Game continues.')
```

Order of operations In terms of order of operations, **and** is done before **or**, so if you have a complicated condition that contains both, you may need parentheses around the **or** condition. Think of **and** as being like multiplication and **or** as being like addition. Here is an example:

```
if (score<1000 or time>20) and turns_remaining==0:
    print('Game over.')
```

4.3 Common Mistakes

Mistake 1 The operator for equality consists of two equals signs. It is a really common error to forget one of the equals signs.

Incorrect Correct

```
if x=1: if x==1:
```

Mistake 2 A common mistake is to use **and** where **or** is needed or vice-versa. Consider the following if statements:

```
if x>1 and x<100: if x>1 or x<100:
```

4.4. ELIF 29

The first statement is the correct one. If x is any value between 1 and 100, then the statement will be true. The idea is that x has to be *both* greater than 1 *and* less than 100. On the other hand, the second statement is not what we want because for it to be true, *either* x has to be greater than 1 *or* x has to be less than 100. But every number satisfies this. The lesson here is if your program is not working correctly, check your **and**'s and **or**'s.

Mistake 3 Another very common mistake is to write something like below: **if** grade>=80 **and** <90:

This will lead to a syntax error. We have to be explicit. The correct statement is **if** grade>=80 **and** grade<90:

On the other hand, there is a nice shortcut that does work in Python (though not in many other programming languages):

```
if 80<=grade<90:  
4.4 elif
```

A simple use of an if statement is to assign letter grades. Suppose that scores 90 and above are A's, scores in the 80s are B's, 70s are C's, 60s are D's, and anything below 60 is an F. Here is one way to do this:

```
grade = eval(input('Enter your score: '))  
if grade>=90:  
    print('A')  
if grade>=80 and grade<90: print('B')  
if grade>=70 and grade<80: print('C')  
if grade>=60 and grade<70:  
    print('D')  
if grade<60: print('F')
```

The code above is pretty straightforward and it works. However, a more elegant way to do it is shown below.

```
grade = eval(input('Enter your score: '))  
if grade>=90: print('A')  
elif grade>=80: print('B')  
elif grade>=70: print('C')  
elif grade>=60:  
    print('D')  
  
else:  
    print('F')
```

With the separate if statements, each condition is checked regardless of whether it really needs to be. That is, if the score is a 95, the first program will print an A but then continue on and check to see if the score is a B, C, etc., which is a bit of a waste. Using **elif**, as soon as we find where the score matches, we stop checking conditions and skip all the way to the end of the whole block of statements. An added benefit of this is that the conditions we use in the **elif** statements are simpler than in their **if** counterparts. For instance, when using **elif**, the second part of the second if statement condition, `grade<90`, becomes unnecessary because the corresponding **elif**

does not have to worry about a score of 90 or above, as such a score would have already been caught by the first if statement.

You can get along just fine without **elif**, but it can often make your code simpler.

4.5 Exercises

Write a program that asks the user to enter a length in centimeters. If the user enters a negative length, the program should tell the user that the entry is invalid. Otherwise, the program should convert the length to inches and print out the result. There are 2.54 centimeters in an inch.

Ask the user for a temperature. Then ask them what units, Celsius or Fahrenheit, the temperature is in. Your program should convert the temperature to the other



unit. The conversions are $F = C + 32$ and $C = (F - 32)$.

Ask the user to enter a temperature in Celsius. The program should print a message based on the temperature:

If the temperature is less than -273.15, print that the temperature is invalid because it is below absolute zero.

If it is exactly -273.15, print that the temperature is absolute 0.

If the temperature is between -273.15 and 0, print that the temperature is below freezing.

If it is 0, print that the temperature is at the freezing point.

If it is between 0 and 100, print that the temperature is in the normal range. If it is 100, print that the temperature is at the boiling point.

If it is above 100, print that the temperature is above the boiling point.

Write a program that asks the user how many credits they have taken. If they have taken 23 or less, print that the student is a freshman. If they have taken between 24 and 53, print that they are a sophomore. The range for juniors is 54 to 83, and for seniors it is 84 and over.

4.5. EXERCISES 31

Generate a random number between 1 and 10. Ask the user to guess the number and print a message based on whether they get it right or not.

A store charges \$12 per item if you buy less than 10 items. If you buy between 10 and 99 items, the cost is \$10 per item. If you buy 100 or more items, the cost is \$7 per item. Write a program that asks the user how many items they are buying and prints the total cost.

Write a program that asks the user for two numbers and prints Close if the numbers are within .001 of each other and Notclose otherwise.

A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Write a program that asks the user for a year and prints out whether it is a leap year or not.

Write a program that asks the user to enter a number and prints out all the divisors of that number. [Hint: the % operator is used to tell if a number is divisible by something. See Section

[3.2.](#)]

Write a multiplication game program for kids. The program should give the player ten randomly generated multiplication questions to do. After each, the program should tell them whether they got it right or wrong and what the correct answer is.

Question 1: $3 \times 4 = 12$ Right!

Question 2: $8 \times 6 = 44$ Wrong. The answer is 48.

...

...

Question 10: $7 \times 7 = 49$ Right.

Write a program that asks the user for an hour between 1 and 12, asks them to enter am or pm, and asks them how many hours into the future they want to go. Print out what the hour will be that many hours into the future, printing am or pm as appropriate. An example is shown below.

Enter hour: 8 am (1) or pm (2)? 1

How many hours ahead? 5
New hour: 1 pm

A jar of Halloween candy contains an unknown amount of candy and if you can guess exactly how much candy is in the bowl, then you win all the candy. You ask the person in charge the following: If the candy is divided evenly among 5 people, how many pieces would be left over? The answer is 2 pieces. You then ask about dividing the candy evenly among 6 people, and the amount left over is 3 pieces. Finally, you ask about dividing the candy evenly among 7 people, and the amount left over is 2 pieces. By looking at the bowl, you can tell that there are less than 200 pieces. Write a program to determine how many pieces are in the bowl.

Write a program that lets the user play Rock-Paper-Scissors against the computer. There should be five rounds, and after those five rounds, your program should print out who won and lost or that there is a tie.

Miscellaneous Topics I

This chapter consists of a several common techniques and some other useful information.

5.1 Counting

Very often we want our programs to count how many times something happens. For instance, a video game may need to keep track of how many turns a player has used, or a math program may want to count how many numbers have a special property. The key to counting is to use a variable to keep the count.

Example 1 This program gets 10 numbers from the user and counts how many of those numbers are greater than 10.

```
count = 0
for i in range(10):
    num = eval(input('Enter a number: '))
    if num > 10:
        count = count + 1
print('There are', count, 'numbers greater than 10.')
```

Think of the count variable as if we are keeping a tally on a piece of paper. Every time we get a number larger than 10, we add 1 to our tally. In the program, this is accomplished by the line `count=count+1`. The first line of the program, `count=0`, is important. Without it, the Python interpreter would get

to the `count=count+1` line and spit out an error saying something about not knowing what `count` is. This is because the first time the program gets to this line, it tries to do what it says: take the old value of `count`, add 1 to it, and store the result in `count`. But the first time the program gets there, there is no old value of `count` to use, so the Python interpreter doesn't know what to do. To avoid the error, we need to define `count`, and that is what the first

33

line does. We set it to 0 to indicate that at the start of the program no numbers greater than 10 have been found.

Counting is an extremely common thing. The two things involved are:

`count=0`— Start the count at 0.

`count=count+1` — Increase the count by 1.

Example 2 This modification of the previous example counts how many of the numbers the user enters are greater than 10 and also how many are equal to 0. To count two things we use two count variables.

```
count1 = 0 count2 = 0
for i in range(10):
    num = eval(input('Enter a number:'))
    if num>10: count1=count1+1
    if num==0:
        count2=count2+1
print('There are', count1, 'numbers greater than 10.')
print('There are', count2, 'zeroes.')
```

Example 3 Next we have a slightly trickier example. This program counts how many of the squares from 1^2 to 100^2 end in a 4.

```
count = 0
for i in range(1,101):
    if (i**2)%10==4:
        count = count + 1
print(count)
```

A few notes here: First, because of the aforementioned quirk of the `range` function, we need to use `range(1,101)` to loop through the numbers 1 through 100. The looping variable `i` takes on those values, so the squares from 1^2 to 100^2 are represented by `i**2`. Next, to check if a number ends in 4, a nice mathematical trick is to check if it leaves a remainder of 4 when divided by

10. The modulo operator, %, is used to get the remainder.

5.2 Summing

Closely related to counting is summing, where we want to add up a bunch of numbers.

5.3. SWAPPING

Example 1 This program will add up the numbers from 1 to 100. The way this works is that each time we encounter a new number, we add it to our running total, s.

```
s = 0
for i in range(1,101): s = s + i
print('The sum is', s)
```

Example 2 This program that will ask the user for 10 numbers and then computes their average.

```
s = 0
for i in range(10): num = eval(input('Enter a number: '))
s = s + num
print('The average is', s/10)
```

Example 3 A common use for summing is keeping score in a game. Near the beginning of the game we would set the score variable equal to 0. Then when we want to add to the score we would do something like below:

```
score = score + 10
```

5.3 Swapping

Quite often we will want to swap the values of two variables, x and y. It would be tempting to try the following:

```
x = y
y = x
```

But this will not work. Suppose x is 3 and y is 5. The first line will set x to 5, which is good, but then the second line will set y to 5 also because x is now 5. The trick is to use a third variable to save the value of x:

```
hold = x
x = y
y = hold
```

In many programming languages, this is the usual way to swap variables. Python, however, provides a nice shortcut:


```
x,y = y,x
```

We will learn later exactly why this works. For now, feel free to use whichever method you prefer. The latter method, however, has the advantage of being shorter and easier to understand.

5.4 Flag variables

A flag variable can be used to let one part of your program know when something happens in another part of the program. Here is an example that determines if a number is prime.

```
num = eval(input('Enter number: '))
```

```
flag = 0 for i in range(2,num):
```

```
if num%i==0: flag = 1
```

```
if flag==1:
```

```
print('Not prime')
```

```
else:
```

```
print('Prime')
```

Recall that a number is prime if it has no divisors other than 1 and itself. The way the program above works is flag starts off at 0. We then loop from 2 to num-1. If one of those values turns out to be a divisor, then flag gets set to 1. Once the loop is finished, we check to see if the flag got set or not. If it did, we know there was a divisor, and num isn't prime. Otherwise, the number must be prime.

5.5 Maxes and mins

A common programming task is to find the largest or smallest value in a series of values. Here is an example where we ask the user to enter ten positive numbers and then we print the largest one.

```
largest = eval(input('Enter a positive number: ')) for i in range(9): num =
```

```
eval(input('Enter a positive number: ')) if num>largest:
```

```
largest=num
```

```
print('Largest number:', largest)
```

The key here is the variable `largest` that keeps track of the largest number found so far. We start by setting it equal to the user's first number. Then, every time we get a new number from the user, we check to see if the user's number is larger than the current largest value (which is stored in `largest`). If it is, then we set `largest` equal to the user's number.

If, instead, we want the smallest value, the only change necessary is that `>` becomes `<`, though it would also be good to rename the variable `largest` to `smallest`.

Later on, when we get to lists, we will see a shorter way to find the largest and smallest values, but the technique above is useful to know since you may occasionally run into situations where the list way won't do everything you need it to do.

5.6. COMMENTS

5.6 Comments

A comment is a message to someone reading your program. Comments are often used to describe what a section of code does or how it works, especially with tricky sections of code. Comments have no effect on your program.

Single-line comments For a single-line comment, use the `#` character. *# a slightly sneaky way to get two values at once*

`num1, num2 = eval(input('Enter two numbers separated by commas: '))` You can also put comments at the end of a line:

`count = count + 2` *# each divisor contributes two the count* **Multi-line comments** For comments that span several lines, you can use triple quotes.

```
""" Program name: Hello world
Author: Brian Heinold
Date: 1/9/11 """ print('Hello world')
```

One nice use for the triple quotes is to comment out parts of your code. Often you will want to modify your program but don't want to delete your old code in case your changes don't work. You could comment out the old code so that it is still there if you need it, and it will be ignored when your new program is

run. Here is a simple example:

```
""" print('This line and the next are inside a comment.') print('These lines will  
not get executed.')
```

```
""" print('This line is not in a comment and it will be executed.')
```

 5.7 Simple debugging

Here are two simple techniques for figuring out why a program is not working:

Use the Python shell. After your program has run, you can type in the names of your program's variables to inspect their values and see which ones have the values you expect them to have and which don't. You can also use the Shell to type in small sections of your program and see if they are working.

Add print statements to your program. You can add these at any point in your program to see what the values of your variables are. You can also add a print statement to see if a point in your code is even being reached. For instance, if you think you might have an error in

a condition of an if statement, you can put a print statement into the if block to see if the condition is being triggered.

Here is an example from the part of the primes program from earlier in this chapter. We put a print statement into the for loop to see exactly when the flag variable is being set:

```
flag = 0 num = eval(input('Enter number: ')) for i in range(2,num): if  
num%i==0: flag = 1  
print(i, flag)
```

An empty input statement, like below, can be used to pause your program at a specific point:

```
input ()
```

5.8 Example programs

It is a valuable skill is to be able to read code. In this section we will look in

depth at some simple programs and try to understand how they work.

Example 1 The following program prints Hello a random number of times between 5 and 25.

```
from random import randint
rand_num = randint(5,25) for i in range(rand_num):
print('Hello')
```

The first line in the program is the import statement. This just needs to appear once, usually near the beginning of your program. The next line generates a random number between 5 and 25. Then, remember that to repeat something a specified number of times, we use a for loop. To repeat something 50 times, we would use `range(50)` in our for loop. To repeat something 100 times, we would use `range(100)`. To repeat something a random number of times, we can use `range(rand_num)`, where `rand_num` is a variable holding a random number. Although if we want, we can skip the variable and put the `randint` statement directly in the `range` function, as shown below.

```
from random import randint
for i in range(randint(5,25)):
print('Hello')
```

5.8. EXAMPLE PROGRAMS

Example 2 Compare the following two programs. `from random import`

```
randint from random import randint
rand_num = randint(1,5) for i in range(6):
for i in range(6): rand_num = randint(1,5)
print('Hello'*rand_num) print('Hello'*rand_num)
Hello Hello Hello Hello Hello
Hello Hello Hello
Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello
Hello Hello Hello Hello
Hello Hello Hello
```

The only difference between the programs is in the placement of the `rand_num` statement. In the first program, it is located outside of the for loop, and this means that `rand_num` is set once at the beginning of the program and

retains that same value for the life of the program. Thus every print statement will print Hello the same number of times. In the second program, the rand_num statement is within the loop. Right before each print statement, rand_num is assigned a new random number, and so the number of times Hello is printed will vary from line to line.

Example 3 Let us write a program that generates 10000 random numbers between 1 and 100 and counts how many of them are multiples of 12. Here are the things we will need:

Because we are using random numbers, the first line of the program should import the random module.

We will require a for loop to run 10000 times.

Inside the loop, we will need to generate a random number, check to see if it is divisible by 12, and if so, add 1 to the count.

Since we are counting, we will also need to set the count equal to 0 before we start counting.

To check divisibility by 12, we use the modulo, %, operator.

When we put this all together, we get the following:

```
from random import randint
count = 0
for i in range(10000):
    num = randint(1, 100)
    if num%12==0:
        count=count+1
print('Number of multiples of 12:', count)
```

Indentation matters

A common mistake is incorrect indentation. Suppose we take the above and indent the last line. The program will still run, but it won't run as expected.

```
from random import randint
count = 0
for i in range(10000):
    num = randint(1, 100)
    if num%12==0:
        count=count+1
    print('Number of multiples of 12:', count)
```

When we run it, it outputs a whole bunch of numbers. The reason for this is that by indenting the print statement, we have made it a part of the for loop, so the print statement will be executed 10,000 times.

Suppose we indent the print statement one step further, like below. **from**

```

random import randint
count = 0 for i in range(10000): num = randint(1, 100) if num%12==0:
count=count+1 print('Number of multiples of 12:', count)

```

Now, not only is it part of the for loop, but it is also part of the if statement. What will happen is every time we find a new multiple of 12, we will print the count. Neither this, nor the previous example, is what we want. We just want to print the count once at the end of the program, so we don't want the print statement indented at all.

5.9 Exercises

Write a program that counts how many of the squares of the numbers from 1 to 100 end in a

1.

Write a program that counts how many of the squares of the numbers from 1 to 100 end in a 4 and how many end in a 9.

Write a program that asks the user to enter a value n , and then computes $(1 + \frac{1}{2} + \dots + \frac{1}{n}) - \ln(n)$. The $\ln$ function is $\log$ in the math module.

Write a program to compute the sum $1 - 2 + 3 - 4 + \dots + 1999 - 2000$.

5.9. EXERCISES

Write a program that asks the user to enter a number and prints the sum of the divisors of that number. The sum of the divisors of a number is an important function in number theory.

A number is called a *perfect number* if it is equal to the sum of all of its divisors, not including the number itself. For instance, 6 is a perfect number because the divisors of 6 are 1, 2, 3, 6 and $6 = 1 + 2 + 3$. As another example, 28 is a perfect number because its divisors are 1, 2, 4, 7, 14, 28 and $28 = 1 + 2 + 4 + 7 + 14$. However, 15 is not a perfect number because its divisors are 1, 3, 5, 15 and $15 \neq 1 + 3 + 5$. Write a program that finds all four of the perfect numbers that are less than 10000.

An integer is called *squarefree* if it is not divisible by any perfect squares other than 1. For instance, 42 is squarefree because its divisors are 1, 2, 3, 6, 7, 21, and 42, and none of those numbers (except 1) is a perfect square. On the other hand, 45 is not squarefree because it is divisible by 9, which is a

perfect square. Write a program that asks the user for an integer and tells them if it is squarefree or not.

Write a program that swaps the values of three variables x , y , and z , so that x gets the value of y , y gets the value of z , and z gets the value of x . Write a program to count how many integers from 1 to 1000 are not perfect squares, perfect cubes, or perfect fifth powers.

Ask the user to enter 10 test scores. Write a program to do the following:
Print out the highest and lowest scores.
Print out the average of the scores.
Print out the second largest score.

If any of the scores is greater than 100, then after all the scores have been entered, print a message warning the user that a value over 100 has been entered.

Drop the two lowest scores and print out the average of the rest of them.

Write a program that computes the factorial of a number. The factorial, $n!$, of a number n is the product of all the integers between 1 and n , including n . For instance, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. [Hint: Try using a multiplicative equivalent of the summing technique.]

Write a program that asks the user to guess a random number between 1 and 10. If they guess right, they get 10 points added to their score, and they lose 1 point for an incorrect guess. Give the user five numbers to guess and print their score after all the guessing is done.

In the last chapter there was an exercise that asked you to create a multiplication game for kids. Improve your program from that exercise to keep track of the number of right and wrong answers. At the end of the program, print a message that varies depending on how many questions the player got right.

This exercise is about the well-known Monty Hall problem. In the problem, you are a contestant on a game show. The host, Monty Hall, shows you three doors. Behind one of those doors is a prize, and behind the other two doors

are goats. You pick a door. Monty Hall, who knows behind which door the prize lies, then opens up one of the doors that doesn't contain the prize. There are now two doors left, and Monty gives you the opportunity to change your choice. Should you keep the same door, change doors, or does it not matter?

Write a program that simulates playing this game 10000 times and calculates what percentage of the time you would win if you switch and what percentage of the time you would win by not switching.

Try the above but with four doors instead of three. There is still only one prize, and Monty still opens up one door and then gives you the opportunity to switch. Strings

Strings are a data type in Python for dealing with text. Python has a number of powerful features for manipulating strings.

6.1 Basics

Creating a string A string is created by enclosing text in quotes. You can use either single quotes, `'`, or double quotes, `"`. A triple-quote can be used for multiline strings. Here are some examples:

```
s = 'Hello' t = "Hello" m = """This is a long string that is spread across two
lines."""
```

Input Recall from Chapter 1 that when getting numerical input we use an **eval** statement with the **input** statement, but when getting text, we do not use **eval**. The difference is illustrated below:

```
num = eval(input('Enter a number: ')) string = input('Enter a string: ')
```

Empty string The empty string `"` is the string equivalent of the number 0. It is a string with nothing in it. We have seen it before, in the print statement's optional argument, `sep=""`.

Length To get the length of a string (how many characters it has), use the built-in function **len**.

For example, **len**('Hello') is 5.

6.2 Concatenation and repetition

The operators `+` and `*` can be used on strings. The `+` operator combines two strings. This operation is called *concatenation*. The `*` repeats a string a certain number of times. Here are some examples.

Expression Result

`'AB'+'cd'` `'ABcd'`

`'A'+'7'+'B'` `'A7B'`

`'Hi' *4` `'HiHiHiHi'`

Example 1 If we want to print a long row of dashes, we can do the following
`print('-'*75)`

Example 2 The `+` operator can be used to build up a string, piece by piece, analogously to the way we built up counts and sums in Sections 5.1 and 5.2. Here is an example that repeatedly asks the user to enter a letter and builds up a string consisting of only the vowels that the user entered.

```
= " for i in range(10):  
= input('Enter a letter: ') if t=='a' or t=='e' or t=='i' or t=='o'  
or t=='u': s = s + t
```

```
print(s)
```

This technique is very useful.

6.3 The in operator

The `in` operator is used to tell if a string contains something. For example:

```
if 'a' in string:
```

```
print('Your string contains the letter a.')
```

You can combine `in` with the `not` operator to tell if a string does not contain something:

```
if ';' not in string:
```

```
print('Your string does not contain any semicolons.')
```

Example In the previous section we had the long if condition

```
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
```

Using the `in` operator, we can replace that statement with the following:

if t in 'aeiou':

6.4. INDEXING

6.4 Indexing

We will often want to pick out individual characters from a string. Python uses square brackets to do this. The table below gives some examples of indexing the string `s='Python'`.

Statement	Result	Description

<code>s[0]</code>	P	first character of s

<code>s[1]</code>	y	second character of s

<code>s[-1]</code>	n	last character of s

<code>s[-2]</code>	o	second-to-last character of s

The first character of s is `s[0]`, not `s[1]`. Remember that in programming, counting usually starts at 0, not 1.

Negative indices count backwards from the end of the string.

A common error Suppose `s='Python'` and we try to do `s[12]`. There are only six characters in the string and Python will raise the following error message:

IndexError: string index out of range

You *will* see this message again. Remember that it happens when you try to read past the end of a string.

6.5 Slices

A *slice* is used to pick out part of a string. It behaves like a combination of indexing and the `range` function. Below we have some examples with the string `s='abcdefghij'`.

index: 0 1 2 3 4 5 6 7 8 9 letters: a b c d e f g h i j

Code	Result	Description

<code>s[2:5]</code>	cde	characters at indices 2, 3, 4

<code>s[:5]</code>	abcde	first five characters

<code>s[5:]</code>	fghij	characters from index 5 to the end

<code>s[-2:]</code>	ij	last two characters

<code>s[:]</code>	abcdefghij	entire string

<code>s[1:7:2]</code>	bdf	characters from index 1 to 6, by twos

<code>s[::-1]</code>	jihgfedcba	a negative step reverses the

string

The basic structure is *string name*[*starting location* : *ending location*+1]

Slices have the same quirk as the **range** function in that they do not include the ending location. For instance, in the example above, `s[2:5]` gives the characters in indices 2, 3, and 4, but not the character in index 5.

We can leave either the starting or ending locations blank. If we leave the starting location blank, it defaults to the start of the string. So `s[:5]` gives the first five characters of `s`. If we leave the ending location blank, it defaults to the end of the string. So `s[5:]` will give all the characters from index 5 to the end. If we use negative indices, we can get the ending characters of the string. For instance, `s[2:]` gives the last two characters.

There is an optional third argument, just like in the **range** statement, that can specify the step. For example, `s[1:7:2]` steps through the string by twos, selecting the characters at indices 1, 3, and 5 (but not 7, because of the aforementioned quirk). The most useful step is -1, which steps backwards through the string, reversing the order of the characters.

6.6 Changing individual characters of a string

Suppose we have a string called `s` and we want to change the character at index 4 of `s` to **'X'**. It is tempting to try `s[4]='X'`, but that unfortunately will not work. Python strings are *immutable*, which means we can't modify any part of them. There is more on why this is in Section 19.1. If we want to change a character of `s`, we have to instead build a new string from `s` and reassign it to `s`.

Here is code that will change the character at index 4 to **'X'**:

```
s = s[:4] + 'X' + s[5:]
```

The idea of this is we take all the characters up to index 4, then X, and then all of the characters after index 4.

6.7 Looping

Very often we will want to scan through a string one character at a time. A for loop like the one below can be used to do that. It loops through a string called `s`, printing the string, character by character, each on a separate line:

```
for i in range(len(s)):
    print (s[i])
```

In the **range** statement we have **len(s)** that returns how long *s* is. So, if *s* were 5 characters long, this would be like having **range(5)** and the loop variable *i* would run from 0 to 4. This means that *s[i]* will run through the characters of *s*. This way of looping is useful if we need to keep track of our location in the string during the loop.

If we don't need to keep track of our location, then there is a simpler type of loop we can use:

6.8. STRING METHODS

```
for c in s:
    print(c)
```

This loop will step through *s*, character by character, with *c* holding the current character. You can almost read this like an English sentence, "For every character *c* in *s*, print that character."

6.8 String methods

Strings come with a ton of *methods*, functions that return information about the string or return a new string that is a modified version of the original. Here are some of the most useful ones:

Method Description

`lower()` returns a string with every letter of the original in lowercase

`upper()` returns a string with every letter of the original in uppercase

`replace(x,y)` returns a string with every occurrence of *x* replaced by *y*

`count(x)` counts the number of occurrences of *x* in the string

`index(x)` returns the location of the first occurrence of *x*

`isalpha()` returns **True** if every character of the string is a letter

Important note One very important note about `lower`, `upper`, and `replace` is that they do not change the original string. If you want to change a string, *s*, to all lowercase, it is not enough to just use *s.lower()*. You need to do the following:

```
s = s.lower()
```

Short examples Here are some examples of string methods in action:

Statement Description

`print(s.count(" "))` prints the number of spaces in the string

`s=s.upper()` changes the string to all caps

`s=s.replace('Hi','Hello')` replaces each 'Hi' in s with 'Hello' `print(s.index('a'))` prints location of the first 'a' in s

isalpha The `isalpha` method is used to tell if a character is a letter or not. It returns **True** if the character is a letter and **False** otherwise. When used with an entire string, it will only return **True** if every character of the string is a letter. The values **True** and **False** are called booleans and are covered in Section 10.2. For now, though, just remember that you can use `isalpha` in if conditions. Here is a simple example:

```
s = input('Enter a string')
if s[0].isalpha():
    print('Your string starts with a letter')
if not s.isalpha():
    print('Your string contains a non-letter.')
```

A note about index If you try to find the index of something that is not in a string, Python will raise an error. For instance, if `s='abc'` and you try `s.index('z')`, you will get an error. One way around this is to check first, like below:

```
if 'z' in s:
    location = s.index('z')
```

Other string methods There are many more string methods. For instance, there are methods `isdigit` and `isalnum`, which are analogous to `isalpha`. Some other useful methods we will learn about later are `join` and `split`. To see a list of all the string methods, type `dir(str)` into the Python shell. If you do this, you will see a bunch of names that start with `__`. You can ignore them. To read Python's documentation for one of the methods, say the `isdigit` method, type `help(str.isdigit)`.

6.9 Escape characters

The backslash, \, is used to get certain special characters, called escape characters, into your string. There are a variety of escape characters, and here are the most useful ones:

\n the *newline character*. It is used to advance to the next line. Here is an example:

```
print ('Hi\n\nthere!')
```

Hi There!

\' for inserting apostrophes into strings. Say you have the following string: s = 'I can't go'

This will produce an error because the apostrophe will actually end the string. You can use \' to get around this:

```
s = 'I can\'t go'
```

Another option is to use double quotes for the string:

```
"s = I can't go"
```

\" analogous to \'.

6.10. EXAMPLES

\\ This is used to get the backslash itself. For example:

```
filename = 'c:\\programs\\file.py'
```

\t the tab character

6.10 Examples

Example 1 An easy way to print a blank line is `print()`. However, if we want to print ten blank lines, a quick way to do that is the following:

```
print ('\n'*9)
```

Note that we get one of the ten lines from the `print` function itself.

Example 2 Write a program that asks the user for a string and prints out the location of each 'a' in the string.

```
s = input('Enter some text: ')
for i in range(len(s)):
    if s[i]=='a': print(i)
```

We use a loop to scan through the string one character at a time. The loop variable i keeps track of our location in the string, and s[i] gives the character

at that location. Thus, the third line checks each character to see if it is an 'a', and if so, it will print out i, the location of that 'a'.

Example 3 Write a program that asks the user for a string and creates a new string that doubles each character of the original string. For instance, if the user enters Hello, the output should be HHeelllloo.

```
s = input('Enter some text: ')
doubled_s = ""
```

```
for c in s:
    doubled_s = doubled_s + c*2
```

Here we can use the second type of loop from Section 6.7. The variable c will run through the characters of s. We use the repetition operator, *, to double each character. We build up the string s in the way described at the end of Section 6.2.

Example 4 Write a program that asks a user for their name and prints it in the following funny pattern:

E El Elv Elvi Elvis

We will require a loop because we have to repeatedly print sections of the string, and to print the sections of the string, we will use a slice:

```
name = input('Enter your name: ')
for i in range(len(name)):
    print(name[:i+1], end=' ')
```

The one trick is to use the loop variable i in the slice. Since the number of characters we need to print is changing, we need a variable amount in the slice. This is reminiscent of the triangle program from Section 2.4. We want to print one character of the name the first time through the loop, two characters the second time, etc. The loop variable, i, starts at 0 the first time through the loop, then increases to 1 the second time through the loop, etc. Thus we use name[:i+1] to print the first i+1 characters of the name. Finally, to get all the slices to print on the same line, we use the print function's optional argument end="".

Example 5 Write a program that removes all capitalization and common punctuation from a string s.

```
s = s.lower()
for c in '.,;:-?!()\'": s = s.replace(c, "")
```

The way this works is for every character in the string of punctuation, we replace every occurrence of it in `s` with the empty string, `''`. One technical note here: We need the `'` character in a string. As described in the previous section, we get it into the string by using the escape character `\'`.

Example 6 Write a program that, given a string that contains a decimal number, prints out the decimal part of the number. For instance, if given 3.14159, the program should print out .14159.

```
s = input('Enter your decimal number: ') print(s[s.index('.'):])
```

The key here is the `index` method will find where the decimal point is. The decimal part of the number starts there and runs to the end of the string, so we use a slice that starts at `s.index('.')`.

Here is another, more mathematical way, to do this:

```
from math import floor num = eval(input('Enter your decimal number: '))
print(num - floor(num))
```

One difference between the two methods is the first produces a string, whereas the second produces a number.

Example 7 A simple and very old method of sending secret messages is the substitution cipher. Basically, each letter of the alphabet gets replaced by another letter of the alphabet, say every `a` gets replaced with an `x`, and every `b` gets replaced by a `z`, etc. Write a program to implement this.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz' key =
'xznlwbgjhqdyvtkfuompciasr'
secret_message = input('Enter your message: ') secret_message =
secret_message.lower()
```

```
for c in secret_message:
    if c.isalpha():
        print(key[alphabet.index(c)],end="")
    else:
        print(c, end="")
```

The string `key` is a random reordering of the alphabet.

The only tricky part of the program is the for loop. What it does is go through the message one character at a time, and, for every letter it finds, it replaces it with the corresponding letter from the key. This is accomplished by using the index method to find the position in the alphabet of the current letter and replacing that letter with the letter from the key at that position. All non-letter characters are copied as is. The program uses the isalpha method to tell whether the current character is a letter or not.

The code to decipher a message is nearly the same. Just change `key[alphabet.index(c)]` to `alphabet[key.index(c)]`. Section [19.11](#) provides a different approach to the substitution cipher.

6.11 Exercises

Write a program that asks the user to enter a string. The program should then print the following:

The total number of characters in the string

The string repeated 10 times

The first character of the string (remember that string indices start at 0) The first three characters of the string

The last three characters of the string

The string backwards

The seventh character of the string if the string is long enough and a message otherwise

The string with its first and last characters removed

The string in all caps

The string with every *a* replaced with an *e*

The string with every letter replaced by a space

A simple way to estimate the number of words in a string is to count the number of spaces in the string. Write a program that asks the user for a string and returns an estimate of how many words are in the string.

People often forget closing parentheses when entering formulas. Write a program that asks the user to enter a formula and prints out whether the formula has the same number of opening and closing parentheses.

Write a program that asks the user to enter a word and prints out whether that word contains any vowels.

Write a program that asks the user to enter a string. The program should create a new string called `new_string` from the user's string such that the second character is changed to an asterisk and three exclamation points are attached to the end of the string. Finally, print `new_string`. Typical output is shown below:

Enter your string: Qbert Q*ert!!!

Write a program that asks the user to enter a string `s` and then converts `s` to lowercase, removes all the periods and commas from `s`, and prints the resulting string.

Write a program that asks the user to enter a word and determines whether the word is a palindrome or not. A palindrome is a word that reads the same backwards as forwards.

At a certain school, student email addresses end with `@student.college.edu`, while professor email addresses end with `@prof.college.edu`. Write a program that first asks the user how many email addresses they will be entering, and then has the user enter those addresses. After all the email addresses are entered, the program should print out a message indicating either that all the addresses are student addresses or that there were some professor addresses entered.

Ask the user for a number and then print the following, where the pattern ends at the number that the user enters.

1
2
3
4

Write a program that asks the user to enter a string, then prints out each letter of the string doubled and on a separate line. For instance, if the user entered HEY, the output would be

HH
EE
YY

Write a program that asks the user to enter a word that contains the letter *a*. The program should then print the following two lines: On the first line should be the part of the string up to and including the first *a*, and on the second line should be the rest of the string. Sample output is shown below:

Enter a word: buffalo buffa lo

Write a program that asks the user to enter a word and then capitalizes every other letter of that word. So if the user enters rhinoceros, the program should print rHiNoCeRoS.

Write a program that asks the user to enter two strings of the same length. The program should then check to see if the strings are of the same length. If they are not, the program should print an appropriate message and exit. If they are of the same length, the program should alternate the characters of the two strings. For example, if the user enters abcde and ABCDE the program should print out AaBbCcDdEe.

Write a program that asks the user to enter their name in lowercase and then capitalizes the first letter of each word of their name.

When I was a kid, we used to play this game called *Mad Libs*. The way it worked was a friend would ask me for some words and then insert those words into a story at specific places and read the story. The story would often turn out to be pretty funny with the words I had given since I had no idea what the story was about. The words were usually from a specific category, like a place, an animal, etc.

For this problem you will write a *Mad Libs* program. First, you should make up a story and leave out some words of the story. Your program should ask the user to enter some words and tell them what types of words to enter. Then print the full story along with the inserted words. Here is a small example, but you should use your own (longer) example:

Enter a college class: CALCULUS

Enter an adjective: HAPPY

Enter an activity: PLAY BASKETBALL

CALCULUS class was really HAPPY today. We learned how to PLAY BASKETBALL today in class. I can't wait for tomorrow's class!

Companies often try to personalize their offers to make them more attractive. One simple way to do this is just to insert the person's name at various places in the offer. Of course, companies don't manually type in every person's name; everything is computer-generated. Write a program that asks the user for their name and then generates an offer like the one below. For simplicity's sake, you may assume that the person's first and last names are one word each.

Enter name: George Washington

Dear George Washington,

I am pleased to offer you our new Platinum Plus Rewards card at a special introductory APR of 47.99%. George, an offer like this does not come along every day, so I urge you to call now toll-free at 1-800-314-1592. We cannot offer such a low rate for long, George, so call right away.

Write a program that generates the 26-line block of letters partially shown below. Use a loop containing one or two print statements.

abcdefghijklmnopqrstuvwxyz bcdefghijklmnopqrstuvwxyz
cdefghijklmnopqrstuvwxyzab ...

yzabcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz

The goal of this exercise is to see if you can mimic the behavior of the **in** operator and the count and index methods using only variables, for loops, and if statements.

Without using the **in** operator, write a program that asks the user for a string and a letter and prints out whether or not the letter appears in the string.

Without using the count method, write a program that asks the user for a string and a letter and counts how many occurrences there are of the letter in

the string.

Without using the index method, write a program that asks the user for a string and a letter and prints out the index of the first occurrence of the letter in the string. If the letter is not in the string, the program should say so.

Write a program that asks the user for a large integer and inserts commas into it according to the standard American convention for commas in large numbers. For instance, if the user enters 1000000, the output should be 1,000,000.

Write a program that converts a time from one time zone to another. The user enters the time in the usual American way, such as 3:48pm or 11:26am. The first time zone the user enters is that of the original time and the second is the desired time zone. The possible time zones are Eastern, Central, Mountain, or Pacific.

Time: 11:48pm
Starting zone: Pacific
Ending zone: Eastern
2:48am

An anagram of a word is a word that is created by rearranging the letters of the original. For instance, two anagrams of *idle* are *deli* and *lied*. Finding anagrams that are real words is beyond our reach until Chapter [12](#). Instead, write a program that asks the user for a string and returns a random anagram of the string—in other words, a random rearrangement of the letters of that string. A simple way of encrypting a message is to rearrange its characters. One way to rearrange the characters is to pick out the characters at even indices, put them first in the encrypted string, and follow them by the odd characters. For example, the string *message* would be encrypted as *msaesg* because the even characters are *m*, *s*, *a*, *e* (at indices 0, 2, 4, and 6) and the odd characters are *e*, *s*, *g* (at indices 1, 3, and 5).

Write a program that asks the user for a string and uses this method to encrypt the string. (b) Write a program that decrypts a string that was encrypted with this method.

A more general version of the above technique is the *rail fence cipher*, where instead of breaking things into evens and odds, they are broken up by threes, fours or something larger. For instance, in the case of threes, the string *secret message* would be broken into three groups. The first group is *sr sg*, the characters at indices 0, 3, 6, 9 and 12. The second group is *eemse*, the characters at indices 1, 4, 7, 10, and 13. The last group is *ctea*, the characters at indices 2, 5, 8, and 11. The encrypted message is *sr sgeemsectea*.

Write a program that asks the user for a string and uses the rail fence cipher in the threes case to encrypt the string.

Write a decryption program for the threes case.

Write a program that asks the user for a string, and an integer determining whether to break things up by threes, fours, or whatever. Encrypt the string using the rail-fence cipher.

Write a decryption program for the general case.

In calculus, the derivative of x^4 is $4x^3$. The derivative of x^5 is $5x^4$. The derivative of x^6 is $6x^5$. This pattern continues. Write a program that asks the user for input like x^3 or x^{25} and prints the derivative. For example, if the user enters x^3 , the program should print out $3x^2$.

In algebraic expressions, the symbol for multiplication is often left out, as in $3x+4y$ or $3(x+5)$.

Computers prefer those expressions to include the multiplication symbol, like $3*x+4*y$ or $3*(x+5)$. Write a program that asks the user for an algebraic expression and then inserts multiplication symbols where appropriate.

Lists

Say we need to get thirty test scores from a user and do something with them, like put them in order. We could create thirty variables, `score1`, `score2`, ..., `score30`, but that would be very tedious. To then put the scores in order would be extremely difficult. The solution is to use lists.

7.1 Basics

Creating lists Here is a simple list:

```
L = [1,2,3]
```

Use square brackets to indicate the start and end of the list, and separate the items by commas.

The empty list The empty list is []. It is the list equivalent of 0 or ". **Long lists** If you have a long list to enter, you can split it across several lines, like below:

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40]
```

Input We can use `eval(input())` to allow the user to enter a list. Here is an example:

```
L = eval(input('Enter a list: ')) print('The first element is ', L[0])
Enter a list: [5,7,9]
The first element is 5
57
```

Printing lists You can use the `print` function to print the entire contents of a list.

```
L = [1,2,3] print(L)
[1, 2, 3]
```

Data types Lists can contain all kinds of things, even other lists. For example, the following is a valid list:

```
[1, 2.718, 'abc', [5,6,7]]
```

7.2 Similarities to strings

There are a number of things which work the same way for lists as for strings. **len** — The number of items in L is given by `len(L)`.

in — The `in` operator tells you if a list contains something. Here are some examples:

```
if 2 in L:
```

```
print('Your list contains the number 2.')
if 0 not in L:
    print('Your list has no zeroes.')
```

Indexing and slicing— These work exactly as with strings. For example, `L[0]` is the first item of the list `L` and `L[:3]` gives the first three items.

index and count — These methods work the same as they do for strings. `+`

and `*` — The `+` operator adds one list to the end of another. The `*` operator repeats a list. Here are some examples:

Expression	Result
<code>[7,8]+[3,4,5]</code>	<code>[7,8,3,4,5]</code>
<code>[7,8]*3</code>	<code>[7,8,7,8,7,8]</code>

<code>[0]*5</code>	<code>[0,0,0,0,0]</code>

The last example is particularly useful for quickly creating a list of zeroes.

Looping — The same two types of loops that work for strings also work for lists. Both of the following examples print out the items of a list, one-by-one, on separate lines.

```
for i in range(len(L)):
    for item in L:
        print(L[i])
        print(item)
```

The left loop is useful for problems where you need to use the loop variable `i` to keep track of where you are in the loop. If that is not needed, then use the right loop, as it is a little simpler.

7.3. BUILT-IN FUNCTIONS

7.3 Built-in functions

There are several built-in functions that operate on lists. Here are some useful ones:

Function	Description

len	returns the number of items in the list

sum	returns the sum of the items in the list

min	returns the minimum of the items in the list

max	returns the maximum of the items in the list

For example, the following computes the average of the values in a list L:
average = `sum(L)/len(L)`

7.4 List methods

Here are some list methods:

Method Description

`append(x)` adds x to the end of the list

`sort()` sorts the list

`count(x)` returns the number of times x occurs in the list

`index(x)` returns the location of the first occurrence of x

`reverse()` reverses the list

`remove(x)` removes first occurrence of x from the list

`pop(p)` removes the item at index p and returns its value

`insert(p,x)` inserts x at index p of the list

Important note There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list L, just use `L.sort()` and not `L=L.sort()`. In fact, the latter will not work at all.

wrong *right*

`s.replace('X','x')` `s=s.replace('X','x')`

`L=L.sort()` `L.sort()`

Other list methods There are a few others list methods. Type `help(list)` in the Python shell to see some documentation for them.

7.5 Miscellaneous

Making copies of lists Making copies of lists is a little tricky due to the way Python handles lists. Say we have a list L and we want to make a copy of the list and call it M. The expression `M=L` will not work for reasons covered in Section 19.1. For now, do the following in place of `M=L`:

```
M = L[:]
```

Changing lists Changing a specific item in a list is easier than with strings. To change the value in location 2 of L to 100, we simply say `L[2]=100`. If we want to insert the value 100 into location 2 without overwriting what is currently there, we can use the insert method. To delete an entry from a list,

we can use the **del** operator. Some examples are shown below. Assume L=[6,7,8] for each operation.

Operation New L Description

L[1]=9 [6,9,8] replace item at index 1 with 9

L.insert(1,9) [6,9,7,8] insert a 9 at index 1 without replacing

del L[1] [6,8] delete second item

del L[:2] [8] 7.6 Examples

delete first two items **Example 1** Write a program that generates a list L of 50 random numbers between 1 and 100.

```
from random import randint
```

```
L = []
```

```
for i in range(50):
```

```
L.append(randint(1,100))
```

We use the append method to build up the list one item at a time starting with the empty list, []. An alternative to append is to use the following:

```
L = L + [randint(1,100)]
```

Example 2 Replace each element in a list L with its square.

```
for i in range(len(L)):
```

```
L[i] = L[i]**2
```

Example 3 Count how many items in a list L are greater than 50. 7.6.

EXAMPLES

```
count = 0 for item in L:
```

```
if item>50:
```

```
count=count+1
```

Example 4 Given a list L that contains numbers between 1 and 100, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

```
frequencies = [] for i in range(1,101):
```

```
frequencies.append(L.count(i))
```

The key is the list method count that tells how many times a something occurs in a list.

Example 5 Write a program that prints out the two largest and two smallest elements of a list called scores.

```
scores.sort() print('Two smallest: ', scores[0], scores[1]) print('Two largest: ', scores[-1], scores[-2])
```

Once we sort the list, the smallest values are at the beginning and the largest are at the end.

Example 6 Here is a program to play a simple quiz game.

```
num_right = 0
```

```
# Question 1 print('What is the capital of France?', end=' ') guess = input() if guess.lower()=='paris':
```

```
    print ('Correct!') num_right+=1
```

```
else:
```

```
    print('Wrong. The answer is Paris.')
```

```
    print('You have', num_right, 'out of 1 right')
```

```
#Question 2 print('Which state has only one neighbor?', end=' ') guess = input() if guess.lower()=='maine':
```

```
    print ('Correct!') num_right+=1
```

```
else:
```

```
    print('Wrong. The answer is Maine.')
```

```
    print('You have', num_right, 'out of 2 right,')
```

The code works, but it is very tedious. If we want to add more questions, we have to copy and paste one of these blocks of code and then change a bunch of things. If we decide to change one of the questions or the order of the questions, then there is a fair amount of rewriting involved. If we decide to change the design of the game, like not telling the user the correct answer, then every single block of code has to be rewritten. Tedious code like this can often be greatly simplified with lists and loops:

```
questions = ['What is the capital of France?',
```

```
'Which state has only one neighbor?' ]
```

```
answers = ['Paris','Maine']
```

```
num_right = 0 for i in range(len(questions)):
```

```
    guess = input(questions[i]) if guess.lower()==answers[i].lower():
```

```
print('Correct') num_right=num_right+1
else:
print('Wrong. The answer is', answers[i])
print('You have', num_right, 'out of', i+1, 'right.')
```

If you look carefully at this code, you will see that the code in the loop is the nearly the same as the code of one of the blocks in the previous program, except that in the statements where we print the questions and answers, we use `questions[i]` and `answers[i]` in place of the actual text of the questions themselves.

This illustrates the general technique: If you find yourself repeating the same code over and over, try lists and a for loop. The few parts of your repetitious code that are varying are where the list code will go.

The benefits of this are that to change a question, add a question, or change the order, only the questions and answers lists need to be changed. Also, if you want to make a change to the program, like not telling the user the correct answer, then all you have to do is modify a single line, instead of twenty copies of that line spread throughout the program.

7.7 Exercises

Write a program that asks the user to enter a list of integers. Do the following: Print the total number of items in the list.

Print the last item in the list.

Print the list in reverse order.

Print Yes if the list contains a 5 and No otherwise.

Print the number of fives in the list.

Remove the first and last items from the list, sort the remaining items, and print the result.

7.7. EXERCISES

(g) Print how many integers in the list are less than 5.

Write a program that generates a list of 20 random numbers between 1 and 100. Print the list.

Print the average of the elements in the list.

Print the largest and smallest values in the list.

Print the second largest and second smallest entries in the list (e) Print how many even numbers are in the list.

Start with the list [8,9,10]. Do the following:

Set the second entry (index 1) to 17

Add 4, 5, and 6 to the end of the list

Remove the first entry from the list

Sort the list

Double the list

Insert 25 at index 3

The final list should equal [4,5,6,25,10,17,4,5,6,10,17]

Ask the user to enter a list containing numbers between 1 and 12. Then replace all of the entries in the list that are greater than 10 with 10.

Ask the user to enter a list of strings. Create a new list that consists of those strings with their first characters removed.

Create the following lists using a for loop.

A list consisting of the integers 0 through 49

A list containing the squares of the integers 1 through 50.

The list ['a','bb','ccc','dddd', ...] that ends with 26 copies of the letter z.

Write a program that takes any two lists L and M of the same size and adds their elements together to form a new list N whose elements are sums of the corresponding elements in L and M. For instance, if L=[3,1,4] and M=[1,5,9], then N should equal [4,6,13].

Write a program that asks the user for an integer and creates a list that consists of the factors of that integer.

When playing games where you have to roll two dice, it is nice to know the odds of each roll. For instance, the odds of rolling a 12 are about 3%, and the odds of rolling a 7 are about 17%. You can compute these mathematically, but if you don't know the math, you can write a program to do it. To do this, your program should simulate rolling two dice about 10,000 times and compute and print out the percentage of rolls that come out to be 2, 3, 4, ..., 12.

Write a program that rotates the elements of a list so that the element at the first index moves to the second index, the element in the second index moves to the third index, etc., and the element in the last index moves to the first index.

Using a for loop, create the list below, which consists of ones separated by increasingly many zeroes. The last two ones in the list should be separated by ten zeroes.

[1,1,0,1,0,0,1,0,0,0,1,0,0,0,0,1,...]

Write a program that generates 100 random integers that are either 0 or 1. Then find the longest *run* of zeros, the largest number of zeros in a row. For instance, the longest run of zeros in [1,0,1,1,0,0,0,0,1,0,0] is 4.

Write a program that removes any repeated items from a list so that each item appears at most once. For instance, the list [1,1,2,3,4,3,0,0] would become [1,2,3,4,0].

Write a program that asks the user to enter a length in feet. The program should then give the user the option to convert from feet into inches, yards, miles, millimeters, centimeters, meters, or kilometers. Say if the user enters a 1, then the program converts to inches, if they enter a 2, then the program converts to yards, etc. While this can be done with if statements, it is much shorter with lists and it is also easier to add new conversions if you use lists.

There is a provably unbreakable cipher called a one-time pad. The way it works is you shift each character of the message by a random amount between 1 and 26 characters, wrapping around the alphabet if necessary. For instance, if the current character is *y* and the shift is 5, then the new character is *d*. Each character gets its own shift, so there needs to be as many random shifts as there are characters in the message. As an example, suppose the user enters *secret*. The program should generate a random shift between 1 and 26 for each character. Suppose the randomly generated shifts are 1, 3, 2, 10, 8, and 2. The encrypted message would be *thebm*v.

Write a program that asks the user for a message and encrypts the message using the one-time pad. First convert the string to lowercase. Any spaces and

punctuation in the string should be left unchanged. For example, Secret!!! becomes thebmvmv!!! using the shifts above.

Write a program to decrypt a string encrypted as above.

The reason it is called a one-time-pad is that the list of random shifts should only be used once. It becomes easily breakable if the same random shifts are used for more than one message. Moreover, it is only provably unbreakable if the random numbers are truly random, and the numbers generated by randint are not truly random. For this problem, just use randint, but for cryptographically safe random numbers, see Section [22.8](#).

More with Lists

8.1 Lists and the random module

There are some nice functions in the random module that work on lists.

Function Description

choice(L) picks a random item from L

sample(L,n) picks a group of n random items from L

shuffle(L) Shuffles the items of L **Note** The shuffle function modifies the original list, so if you don't want your list changed, you'll need to make a copy of it.

Example 1 We can use choice to pick a name from a list of names. `from random import choice` names = ['Joe', 'Bob', 'Sue', 'Sally'] current_player = choice(names)

Example 2 The sample function is similar to choice. Whereas choice picks one item from a list, sample can be used to pick several.

`from random import sample` names = ['Joe', 'Bob', 'Sue', 'Sally'] team = sample(names, 2)

65

Example 3 The choice function also works with strings, picking a random character from a string. Here is an example that uses choice to fill the screen with a bunch of random characters.

```
from random import choice
s='abcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*()'
for i in range(10000):
    print(choice(s), end="")
```

Example 4 Here is a nice use of shuffle to pick a random ordering of players in a game.

```
from random import shuffle
players = ['Joe', 'Bob', 'Sue', 'Sally']
shuffle(players)
for p in players:
```

```
    print(p, 'it is your turn.')
```

code to play the game goes here...

Example 5 Here we use shuffle divide a group of people into teams of two. Assume we are given a list called names.

```
shuffle(names)
teams = []
for i in range(0, len(names), 2):
    teams.append([names[i], names[i+1]])
```

Each item in teams is a list of two names. The way the code works is we shuffle the names so they are in a random order. The first two names in the shuffled list become the first team, the next two names become the second team, etc. Notice that we use the optional third argument to **range** to skip ahead by two through the list of names.

8.2 split

The split method returns a list of the words of a string. The method assumes that words are separated by whitespace, which can be either spaces, tabs or newline characters. Here is an example:

```
s = 'Hi! This is a test.'
print(s.split())
['Hi!', 'This', 'is', 'a', 'test.']
```

As we can see, since split breaks up the string at spaces, the punctuation will be part of the words. There is a module called string that contains, among other things, a string variable called punctuation that contains common punctuation. We can remove the punctuation from a string s with the following code:

8.3. JOIN

```
from string import punctuation
for c in punctuation: s = s.replace(c, "")
```

Example Here is a program that counts how many times a certain word

occurs in a string.

from string **import** punctuation

```
s = input('Enter a string: ') for c in punctuation: s = s.replace(c, '') s = s.lower() L = s.split()
```

```
word = input('Enter a word: ') print(word, 'appears', L.count(word), 'times.')
```

Optional argument The split method takes an optional argument that allows it to break the string at places other than spaces. Here is an example:

```
s = '1-800-271-8281' print(s.split('-'))  
['1', '800', '271', '8281']  
8.3 join
```

The join method is in some sense the opposite of split. It is a string method that takes a list of strings and joins them together into a single string. Here are some examples, using the list

```
L=['A','B','C']  
Operation Result  
".join(L) A B C  
".join(L) ABC  
';'.join(L) A, B, C  
'***'.join(L) A***B***C
```

Example Write a program that creates an anagram of a given word. An anagram of a word uses the same letters as the word but in a different order. For instance, two anagrams of the word *there* are *three* and *ether*. Don't worry about whether the anagram is a real word or not.

This sounds like something we could use shuffle for, but shuffle only works with lists. What we need to do is convert our string into a list, use shuffle on it, and then convert the list back into a string. To turn a string *s* into a list, we can use **list(s)**. (See Section 10.1.) To turn the list back into a string, we will use join.

```
from random import shuffle word = input('Enter a word: ')  
letter_list = list(word) shuffle(letter_list) anagram = "".join(letter_list)
```

```
print(anagram)
```

8.4 List comprehensions

List comprehensions are a powerful way to create lists. Here is a simple example:

```
L = [i for i in range(5)]
```

This creates the list [0,1,2,3,4]. Notice that the syntax of a list comprehension is somewhat reminiscent of set notation in mathematics. Here are a couple more examples of list comprehensions.

For these examples, assume the following:

```
string = 'Hello'
```

```
= [1,14,5,9,12]
```

```
= ['one', 'two', 'three', 'four', 'five', 'six']
```

List comprehension Resulting list

```
[0 for i in range(10)] [0,0,0,0,0,0,0,0,0,0]
```

```
[i**2 for i in range(1,8)] [1,4,9,16,25,36,49]
```

```
[i*10 for i in L] [10,140,50,90,120]
```

```
[c*2 for c in string] ['HH','ee','ll','ll','oo']
```

```
[m[0] for m in M] ['o','t','t','f','f','s']
```

```
[i for i in L if i<10] [1,5,9]
```

```
[m[0] for m in M if len(m)==3] ['o','t','s']
```

As we see in the last two examples, we can add an **if** to a list comprehension.

Compare the last example with the long way of building the list:

```
L = [] for m in M: if len(m)==3:
```

```
L.append(m)
```

Multiple fors You can use more than one **for** in a list comprehension: 8.5.

USING LIST COMPREHENSIONS

```
L = [[i,j] for i in range(2) for j in range(2)]
```

```
[[0, 0], [0, 1], [1, 0], [1, 1]]
```

This is the equivalent of the following code:

```
L = []
```

```
for i in range(2):
```

```
for j in range(2):
```

```
L.append([i,j])
```

Here is another example:

```
[[i,j] for i in range(4) for j in range(i)]  
[[1, 0], [2, 0], [2, 1], [3, 0], [3, 1], [3, 2]]
```

8.5 Using list comprehensions

To further demonstrate the power of list comprehensions, we will do the first four examples of Section 7.6 in one line apiece using list comprehensions.

Example 1 Write a program that generates a list L of 50 random numbers between 1 and 100.

```
L = [randint(1,100) for i in range(50)]
```

Example 2 Replace each element in a list L with its square.

```
L = [i**2 for i in L]
```

Example 3 Count how many items in a list L are greater than 50. `len([i for i in L if i>50])`

Example 4 Given a list L that contains numbers between 1 and 100, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

```
frequencies = [L.count(i) for i in range(1,101)]
```

Another example The join method can often be used with list comprehensions to quickly build up a string. Here we create a string that contains a random assortment of 1000 letters.

```
from random import choice alphabet = 'abcdefghijklmnopqrstuvwxyz' s =  
"".join([choice(alphabet) for i in range(1000)])
```

One more example Suppose we have a list whose elements are lists of size 2, like below:

```
= [[1,2], [3,4], [5,6]]
```

If we want to flip the order of the entries in the lists, we can use the following list comprehension:

```
= [[y,x] for x,y in L]  
[[2, 1], [4, 3], [6, 5]]
```

Note You can certainly get away without using list comprehensions, but once you get the hang of them, you'll find they are both quicker to write and easier

to read than the longer ways of creating lists.

8.6 Two-dimensional lists

There are a number of common things that can be represented by twodimensional lists, like a Tictac-toe board or the pixels on a computer screen. In Python, one way to create a two-dimensional list is to create a list whose items are themselves lists. Here is an example:

```
L = [[1,2,3],  
     [4,5,6],  
     [7,8,9]]
```

Indexing We use two indices to access individual items. To get the entry in row *r*, column *c*, use the following:

```
L[r][c]
```

Printing a two-dimensional list To print a two-dimensional list, you can use nested for loops. The following example prints a 10×5 list:

```
for r in range(10):  
    for c in range(5):  
        print(L[r][c], end=" ")  
    print()
```

Another option is to use the pprint function of the pprint module. This function is used to “pretty-print” its argument. Here is an example to print a list *L*: `from pprint import pprint` pprint(*L*)

The pprint function can be used to nicely print ordinary lists and other objects in Python.

Working with two-dimensional lists Nested for loops, like the ones used in printing a twodimensional list, can also be used to process the items in a twodimensional list. Here is an example that counts how many entries in a 10×5 list are even.

8.6. TWO-DIMENSIONAL LISTS

```
count = 0  
for r in range(10):
```

```
for c in range(5):
    if L[r][c]%2==0: count = count + 1
```

This can also be done with a list comprehension:

```
count = sum([1 for r in range(10) for c in range(5) if L[r][c]%2==0])
```

Creating large two-dimensional lists To create a larger list, you can use a list comprehension like below:

```
L = [[0]*50 for i in range(100)]
```

This creates a list of zeroes with 100 rows and 50 columns.

Picking out rows and columns To get the row r of L (starting at row $r = 0$), use the following:

```
L[r]
```

To get the column c of L (starting at column $c = 0$), use a list comprehension:

```
[L[i][c] for i in range(len(L))]
```

Flattening a list To flatten a two-dimensional list, that is, return a onedimensional list of its elements, use the following:

```
[j for row in L for j in row]
```

For instance, suppose we have the following list:

```
L = [[1,2,3],
      [4,5,6],
      [7,8,9]]
```

The flattened list will be:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Higher dimensions Creating and using 3-dimensional and higher lists is similar. Here we create a $5 \times 5 \times 5$ list:

```
L = [[[0]*5 for i in range(5)] for j in range(5)]
```

It is a list whose items are lists of lists. The first entry in the list is

```
L[0][0][0]
```

8.7 Exercises

Write a program that asks the user to enter some text and then counts how many articles are in the text. Articles are the words 'a', 'an', and 'the'.

Write a program that allows the user to enter five numbers (read as strings). Create a string that consists of the user's numbers separated by plus signs. For instance, if the user enters 2, 5, 11, 33, and 55, then the string should be '2+5+11+33+55'.

(a) Ask the user to enter a sentence and print out the third word of the sentence. (b) Ask the user to enter a sentence and print out every third word of the sentence.

(a) Write a program that asks the user to enter a sentence and then randomly rearranges the words of the sentence. Don't worry about getting punctuation or capitalization correct.

(b) Do the above problem, but now make sure that the sentence starts with a capital, that the original first word is not capitalized if it comes in the middle of the sentence, and that the period is in the right place.

Write a simple quote-of-the-day program. The program should contain a list of quotes, and when the user runs the program, a randomly selected quote should be printed.

Write a simple lottery drawing program. The lottery drawing should consist of six different numbers between 1 and 48.

Write a program that estimates the average number of drawings it takes before the user's numbers are picked in a lottery that consists of correctly picking six different numbers that are between 1 and 10. To do this, run a loop 1000 times that randomly generates a set of user numbers and simulates drawings until the user's numbers are drawn. Find the average number of drawings needed over the 1000 times the loop runs.

Write a program that simulates drawing names out of a hat. In this drawing, the number of hat entries each person gets may vary. Allow the user to input a list of names and a list of how many entries each person has in the drawing, and print out who wins the drawing.

Write a simple quiz game that has a list of ten questions and a list of answers to those questions. The game should give the player four randomly selected questions to answer. It should ask the questions one-by-one, and tell the player whether they got the question right or wrong. At the end it should print out how many out of four they got right.

Write a censoring program. Allow the user to enter some text and your program should print out the text with all the curse words starred out. The number of stars should match the length of the curse word. For the purposes of this program, just use the “curse” words *darn*, *dang*, *freakin*, *heck*, and *shoot*. Sample output is below:

Enter some text: Oh shoot, I thought I had the dang problem figured out.
Darn it. Oh well, it was a heck of a freakin try.

Oh *****, I thought I had the **** problem figured out.
**** it. Oh well, it was a **** of a ***** try.

8.7. EXERCISES

Section [8.3](#) described how to use the shuffle method to create a random anagram of a string. Use the choice method to create a random anagram of a string.

Write a program that gets a string from the user containing a potential telephone number. The program should print Valid if it decides the phone number is a real phone number, and Invalid otherwise. A phone number is considered valid as long as it is written in the form *abc-def-hijk* or *1-abc-def-hijk*. The dashes must be included, the phone number should contain only numbers and dashes, and the number of digits in each group must be correct. Test your program with the output shown below.

Enter a phone number: 1-301-447-5820

Valid

Enter a phone number: 301-447-5820

Valid

Enter a phone number: 301-4477-5820

Invalid

Enter a phone number: 3X1-447-5820

Invalid

Enter a phone number: 3014475820 Invalid

Let L be a list of strings. Write list comprehensions that create new lists from L for each of the following.

A list that consists of the strings of s with their first characters removed
A list of the lengths of the strings of s

A list that consists of only those strings of s that are at least three characters long

Use a list comprehension to produce a list that consists of all palindromic numbers between 100 and 1000.

Use a list comprehension to create the list below, which consists of ones separated by increasingly many zeroes. The last two ones in the list should be separated by ten zeroes.

[1,1,0,1,0,0,1,0,0,0,1,0,0,0,0,1,...]

Let $L=[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]$. Use a list comprehension to produce a list of the gaps between consecutive entries in L . Then find the maximum gap size and the percentage of gaps that have size 2.

Write a program that finds the average of all of the entries in a 4×4 list of integers.

Write a program that creates a 10×10 list of random integers between 1 and 100. Then do the following:

Print the list.

Find the largest value in the third row.

Find the smallest value in the sixth column.

Write a program that creates and prints an 8×8 list whose entries alternate between 1 and 2 in a checkerboard pattern, starting with 1 in the upper left corner.

Write a program that checks to see if a 4×4 list is a magic square. In a magic square, every row, column, and the two diagonals add up to the same

value.

Write a program that asks the user to enter a length. The program should ask them what unit the length is in and what unit they would like to convert it to. The possible units are inches, yards, miles, millimeters, centimeters, meters, and kilometers. While this can be done with 25 if statements, it is shorter and easier to add on to if you use a two-dimensional list of conversions, so please use lists for this problem.

The following is useful as part of a program to play *Battleship*. Suppose you have a 5×5 list that consists of zeroes and ones. Ask the user to enter a row and a column. If the entry in the list at that row and column is a one, the program should print Hit and otherwise it should print Miss.

This exercise is useful in creating a *Memory* game. Randomly generate a 6×6 list of assorted characters such that there are exactly two of each character. An example is shown below.

@ 5 # A A !

5 0 b @ \$ z

\$ N x ! N z 0 - + # b :

- : + c c x

The following is useful in implementing computer players in a number of different games. Write a program that creates a 5×5 list consisting of zeroes and ones. Your program should then pick a random location in the list that contains a zero and change it to a one. If all the entries are one, the program should say so. [Hint: one way to do this is to create a new list whose items are the coordinates of all the ones in the list and use the choice method to randomly select one. Use a two-element list to represent a set of coordinates.]

Here is an old puzzle question you can solve with a computer program. There is only one five-digit number n that is such that every one of the following ten numbers shares exactly one digit in common in the same position as n . Find n .

01265,12171,23257,34548,45970,56236,67324,78084,89872,99414 We usually refer to the entries of a two-dimensional list by their row and column,

like below on the left. Another way is shown below on the right.

```
(0,0) (0,1) (0,2) 0 1 2
(1,0) (1,1) (1,2) 3 4 5
(2,0) (2,1) (2,2) 6 7 8
```

Write some code that translates from the left representation to the right one. The // and % operators will be useful. Be sure your code works for arrays of any size.

Write some code that translates from the right representation to the left one. While loops

We have already learned about for loops, which allow us to repeat things a specified number of times. Sometimes, though, we need to repeat something, but we don't know ahead of time exactly how many times it has to be repeated. For instance, a game of Tic-tac-toe keeps going until someone wins or there are no more moves to be made, so the number of turns will vary from game to game. This is a situation that would call for a while loop.

9.1 Examples

Example 1 Let's go back to the first program we wrote back in Section 1.3, the temperature converter. One annoying thing about it is that the user has to restart the program for every new temperature. A while loop will allow the user to repeatedly enter temperatures. A simple way for the user to indicate that they are done is to have them enter a nonsense temperature like -1000 (which is below absolute 0). This is done below:

```
temp = 0 while temp != -1000:
temp = eval(input('Enter a temperature (-1000 to quit): ')) print('In
Fahrenheit that is', 9/5*temp+32)
```

Look at the while statement first. It says that we will keep looping, that is, keep getting and converting temperatures, as long as the temperature entered is not -1000. As soon as -1000 is entered, the while loop stops. Tracing through, the program first compares temp to -1000. If temp is not -1000, then the program asks for a temperature and converts it. The program then

loops back up and again compares temp to -1000. If temp is not -1000, the program will ask for another temperature, convert it, and then loop back up again and do another comparison. It continues this process until the user enters -1000.

We need the line temp=0 at the start, as without it, we would get a name error. The program would

75

get to the **while** statement, try to see if temp is not equal to -1000 and run into a problem because temp doesn't yet exist. To take care of this, we just declare temp equal to 0. There is nothing special about the value 0 here. We could set it to anything except -1000. (Setting it to -1000 would cause the condition on the while loop to be false right from the start and the loop would never run.)

Note that is natural to think of the while loop as continuing looping until the user enters -1000. However, when we construct the condition, instead of thinking about when to stop looping, we instead need to think in terms of what has to be true in order to keep going.

A while loop is a lot like an if statement. The difference is that the indented statements in an if block will only be executed once, whereas the indented statements in a while loop are repeatedly executed.

Example 2 One problem with the previous program is that when the user enters in -1000 to quit, the program still converts the value -1000 and doesn't give any message to indicate that the program has ended. A nicer way to do the program is shown below.

```
temp = 0 while temp != -1000: temp = eval(input('Enter a temperature (-1000 to quit): ')) if temp != -1000:
```

```
print ('In Fahrenheit that is', 9/5*temp+32)
```

```
else:
```

```
print('Bye!')
```

Example 3 When first met if statements in Section 4.1, we wrote a program that played a simple random number guessing game. The problem with that

program is that the player only gets one guess. We can, in a sense, replace the if statement in that program with a while loop to create a program that allows the user to keep guessing until they get it right.

```
from random import randint secret_num = randint(1,10) guess = 0 while  
guess != secret_num:
```

```
guess = eval(input('Guess the secret number: '))  
print('You finally got it!')
```

The condition `guess!=secret_num` says that as long as the current guess is not correct, we will keep looping. In this case, the loop consists of one statement, the input statement, and so the program will keep asking the user for a guess until their guess is correct. We require the line `guess=0` prior to the while loop so that the first time the program reaches the loop, there is something in `guess` for the program to use in the comparison. The exact value of `guess` doesn't really matter at this point. We just want something that is guaranteed to be different than `secret_num`. When the user finally guesses the right answer, the loop ends and program control moves to the `print` statement after the loop, which prints a congratulatory message to the player.

9.1. EXAMPLES

Example 4 We can use a while loop to mimic a for loop, as shown below. Both loops have the exact same effect.

```
for i in range(10): i=0 print(i) while i<10: print(i) i=i+1
```

Remember that the for loop starts with the loop variable `i` equal to 0 and ends with it equal to 9. To use a while loop to mimic the for loop, we have to manually create our own loop variable `i`. We start by setting it to 0. In the while loop we have the same `print` statement as in the for loop, but we have another statement, `i=i+1`, to manually increase the loop variable, something that the for loop does automatically.

Example 5 Below is our old friend that converts from Fahrenheit to Celsius.

```
temp = eval(input('Enter a temperature in Celsius: ')) print('In Fahrenheit,  
that is', 9/5*temp+32)
```

A program that gets input from a user may want to check to see that the user

has entered valid data. The smallest possible temperature is absolute zero, -273.15 °C. The program below takes absolute zero into account:

```
temp = eval(input('Enter a temperature in Celsius: ')) if temp<=-273.15:
print('That temperature is not possible.')
else:
print('In Fahrenheit, that is', 9/5*temp+32)
```

One way to improve this is to allow the user to keep reentering the temperature until they enter a valid one. You may have experienced something similar using an online form to enter a phone number or a credit card number. If you enter an invalid number, you are told to reenter it. In the code below, the while loop acts very similarly to the if statement in the previous example.

```
temp = eval(input('Enter a temperature in Celsius: ')) while temp<=-273.15:
temp = eval(input('Impossible. Enter a valid temperature: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

Note that we do not need an **else** statement here, like we had with the if statement.. The condition on the while loop guarantees that we will only get to the **print** statement once the user enters a valid temperature. Until that point, the program will be stuck in the loop, continually asking the user for a new temperature.

Example 6 As mentioned before, it is a valuable skill is to be able to read code. One way to do so is to pretend to be the Python interpreter and go through the code lineby line. Let's try it with the code below.

```
i = 0 while i<50: print(i) i=i+2
print('Bye!')
```

The variable *i* gets set to 0 to start. Next, the program tests the condition on the while loop. Because *i* is 0, which is less than 50, the code indented under the **while** statement will get executed. This code prints the current value of *i* and then executes the statement *i*=*i*+2 which adds 2 to *i*.

The variable *i* is now 2 and the program loops back to the **while** statement. It checks to see if *i* is less than 50, and since *i* is 2, which is less than 50, the

indented code should be executed again. So we print *i* again, add 2 to it, and then loop back to check the while loop condition again. We keep doing this until finally *i* gets to 50.

At this point, the **while** condition will finally not be true anymore and the program jumps down to the first statement after the **while**, which prints Bye!. The end result of the program is the numbers 0, 2, 4, ..., 48 followed by the message, Bye!.

9.2 Infinite loops

When working with while loops, sooner or later you will accidentally send Python into a neverending loop. Here is an example:

```
i=0 while i<10: print(i)
```

In this program, the value of *i* never changes and so the condition *i*<10 is always true. Python will continuously print zeroes. To stop a program caught in a neverending loop, use Restart Shell under the Shell menu. You can use this to stop a Python program before it is finished executing.

Sometimes a never-ending loop is what you want. A simple way to create one is shown below:

```
while True:
```

```
# statements to be repeated go here
```

The value **True** is called a boolean value and is discussed further in Section 10.2.

9.3 The break statement

The **break** statement can be used to break out of a for or while loop before the loop is finished.

9.4. THE ELSE STATEMENT

Example 1 Here is a program that allows the user to enter up to 10 numbers. The user can stop early by entering a negative number.

```
for i in range(10):
```

```
    num = eval(input('Enter number: ')) if num<0: break
```

This could also be accomplished with a while loop.

```
i=0 num=1 while i<10 and num>0:
```

```
    num = eval(input('Enter a number: '))
```

Either method is ok. In many cases the **break** statement can help make your code easier to understand and less clumsy.

Example 2 Earlier in the chapter, we used a while loop to allow the user to repeatedly enter temperatures to be converted. Here is, more or less, the original version on the left compared with a different approach using the **break** statement.

```
temp = 0 while True:  
while temp!=-1000: temp = eval(input(': '))  
temp = eval(input(': ')) if temp== -1000:  
if temp!=-1000: print('Bye')  
print(9/5*temp+32) break  
else: print(9/5*temp+32)  
print('Bye!')
```

9.4 The else statement

There is an optional **else** that you can use with **break** statements. The code indented under the **else** gets executed only if the loop completes without a **break** happening.

Example 1 This is a simple example based off of Example 1 of the previous section.

```
for i in range(10):  
num = eval(input('Enter number: ')) if num<0:  
print('Stopped early') break  
else:  
print('User entered all ten values')
```

The program allows the user to enter up to 10 numbers. If they enter a negative, then the program prints Stopped early and asks for no more numbers. If the user enters no negatives, then the program prints User entered all ten values.

Example 2 Here are two ways to check if an integer num is prime. A prime number is a number whose only divisors are 1 and itself. The approach on the

left uses a while loop, while the approach on the right uses a for/break loop:

```
i=2 for i in range(2, num):  
while i<num and num%i!=0: if num%i==0:  
i=i+1 print('Not prime')  
if i==num: break  
print('Prime') else:  
else: print('Prime')  
print('Not prime')
```

The idea behind both approaches is to scan through all the integers between 2 and num-1, and if any of them is a divisor, then we know num is not prime. To see if a value i is a divisor of num, we just have to check to see if num%i is 0.

The idea of the while loop version is we continue looping as long as we haven't found a divisor. If we get all the way through the loop without finding a divisor, then i will equal num, and in that case the number must be prime.

The idea of the for/break version is we loop through all the potential divisors, and as soon as we find one, we know the number is not prime and we print Not prime and stop looping. If we get all the way through the loop without breaking, then we have not found a divisor. In that case the **else** block will execute and print that the number is prime.

9.5 The guessing game, more nicely done

It is worth going through step-by-step how to develop a program. We will modify the guessing game program from Section 9.1 to do the following:

The player only gets five turns.

The program tells the player after each guess if the number is higher or lower.

The program prints appropriate messages for when the player wins and loses.

Below is what we want the program to look like:

Enter your guess (1-100): 50 LOWER. 4 guesses left.

Enter your guess (1-100): 25

9.5. THE GUESSING GAME, MORE NICELY DONE

LOWER. 3 guesses left.

Enter your guess (1-100): 12 LOWER. 2 guesses left.

Enter your guess (1-100): 6 HIGHER. 1 guesses left.

Enter your guess (1-100): 9 LOWER. 0 guesses left.

You lose. The correct number is 8

First, think about what we will need in the program:

We need random numbers, so there will be an import statement at the beginning of the program and a randint function somewhere else.

To allow the user to guess until they either guess right or run out of turns, one solution is to use while loop with a condition that takes care of both of these possibilities.

There will be an input statement to get the user's guess. As this is something that is repeatedly done, it will go inside the loop.

There will be an if statement to take care of the higher/lower thing. As this comparison will be done repeatedly and will depend on the user's guesses, it will go in the loop after the input statement.

There will be a counting variable to keep track of how many turns the player has taken. Each time the user makes a guess, the count will go up by one, so this statement will also go inside the loop.

Next start coding those things that are easy to do:

```
secret_num = randint(1,100) num_guesses = 0
```

```
while #some condition goes here# guess = eval(input('Enter your guess (1-100): ')) num_guesses = num_guesses + 1 # higher/lower if statement goes here
```

For the while loop, we want to continue looping as long as the user has not guessed the secret number and as long as the player has not used up all of their guesses:

```
while guess != secret_num and num_guesses <= 4:
```

The higher/lower if statement can be done like this:

```
if guess < secret_num: print('HIGHER.', 5-num_guesses, 'guesses left.\n')
```

```
elif guess > secret_num:
print('LOWER.', 5-num_guesses, 'guesses left.\n')
else:
print('You got it!')
```

Finally, it would be nice to have a message for the player if they run out of turns. When they run out of turns, the while loop will stop looping and program control will shift to whatever comes outside of the loop. At this point we can print the message, but we only want to do so if the reason that the loop stopped is because of the player running out of turns and not because they guessed correctly. We can accomplish this with an if statement after the loop. This is shown below along with the rest of the completed program.

```
secret_num = randint(1,100) num_guesses = 0 guess = 0
```

```
while guess != secret_num and num_guesses <= 4:
```

```
guess = eval(input('Enter your guess (1-100): ')) num_guesses =
num_guesses + 1 if guess < secret_num: print('HIGHER.', 5-num_guesses,
'guesses left.\n')
```

```
elif guess > secret_num:
print('LOWER.', 5-num_guesses, 'guesses left.\n')
else:
print('You got it!')
if num_guesses==5 and guess != secret_num:
print('You lose. The correct number is', secret_num)
```

Here is an alternative solution using a for/break loop:

```
from random import randint secret_num = randint(1,100)
```

```
for num_guesses in range(5): guess = eval(input('( 100): ')) if guess <
secret_num: print('HIGHER.', 5-num_guesses, 'guesses left.\n')
```

```
elif guess > secret_num:
print('LOWER.', 5-num_guesses, 'guesses left.\n')
else:
print('You got it!') break
else:
```

```
print('You lose. The correct number is', secret_num)
```

9.6. EXERCISES

9.6 Exercises

The code below prints the numbers from 1 to 50. Rewrite the code using a while loop to accomplish the same thing.

```
for i in range(1,51): print(i)
```

(a) Write a program that uses a while loop (not a for loop) to read through a string and print the characters of the string one-by-one on separate lines.

(b) Modify the program above to print out every second character of the string.

A good program will make sure that the data its users enter is valid. Write a program that asks the user for a weight and converts it from kilograms to pounds. Whenever the user enters a weight below 0, the program should tell them that their entry is invalid and then ask them again to enter a weight. [Hint: Use a while loop, not an if statement].

Write a program that asks the user to enter a password. If the user enters the right password, the program should tell them they are logged in to the system. Otherwise, the program should ask them to reenter the password. The user should only get five tries to enter the password, after which point the program should tell them that they are kicked off of the system.

Write a program that allows the user to enter any number of test scores. The user indicates they are done by entering in a negative number. Print how many of the scores are A's (90 or above). Also print out the average.

Modify the higher/lower program so that when there is only one guess left, it says 1 guess, not 1 guesses.

Recall that, given a string `s`, `s.index('x')` returns the index of the first `x` in `s` and an error if there is no `x`.

Write a program that asks the user for a string and a letter. Using a while loop, the program should print the index of the first occurrence of that letter

and a message if the string does not contain the letter.

Write the above program using a for/break loop instead of a while loop.

The GCD (greatest common divisor) of two numbers is the largest number that both are divisible by. For instance, $\text{gcd}(18,42)$ is 6 because the largest number that both 18 and 42 are divisible by is 6. Write a program that asks the user for two numbers and computes their gcd. Shown below is a way to compute the GCD, called Euclid's Algorithm.

First compute the remainder of dividing the larger number by the smaller number. Next, replace the larger number with the smaller number and the smaller number with the remainder.

Repeat this process until the smaller number is 0. The GCD is the last value of the larger number.

A 4000-year old method to compute the square root of 5 is as follows: Start with an initial guess, say 1. Then compute

$$\frac{1}{2} \left(1 + \frac{5}{1} \right) = 3.$$

Next, take that 3 and replace the 1's in the previous formula with 3's. This gives

$$\frac{1}{2} \left(3 + \frac{5}{3} \right)$$

Next replace the 3 in the previous formula with $7/3$. This gives

$$\frac{1}{2} \left(\frac{7}{3} + \frac{5}{7/3} \right)$$

If you keep doing this process of computing the formula, getting a result, and plugging it back in, the values will eventually get closer and closer to $\sqrt{5}$.

This method works for numbers other than 5. Write a program that asks the user for a number and uses this method to estimate the square root of the number correct to within 10^{-10} . The estimate will be correct to within 10^{-10} when the absolute value of the difference between consecutive values is less than 10^{-10} .

Write a program that has a list of ten words, some of which have repeated letters and some which don't. Write a program that picks a random word from the list that does not have any repeated letters.

Write a program that starts with an 5×5 list of zeroes and randomly changes

exactly ten of those zeroes to ones.

Write a program in which you have a list that contains seven integers that can be 0 or 1. Find the first nonzero entry in the list and change it to a 1. If there are no nonzero entries, print a message saying so.

In Chapter 4 there was a problem that asked you to write a program that lets the user play Rock-Paper-Scissors against the computer. In that program there were exactly five rounds. Rewrite the program so that it is a best 3 out of 5. That is, the first player to win three times is the winner.

Write a program to play the following simple game. The player starts with \$100. On each turn a coin is flipped and the player has to guess heads or tails. The player wins \$9 for each correct guess and loses \$10 for each incorrect guess. The game ends either when the player runs out of money or gets to \$200.

Write a program to play the following game. There is a list of several country names and the program randomly picks one. The player then has to guess letters in the word one at a time. Before each guess the country name is displayed with correctly guessed letters filled in and the rest of the letters represented with dashes. For instance, if the country is *Canada* and the player has correctly guessed *a*, *d*, and *n*, the program would display *-ana-da*. The program should continue until the player either guesses all of the letters of the word or gets five letters wrong.

9.6. EXERCISES

Write a text-based version of the game *Memory*. The game should generate a 5×5 board (see the exercise from Chapter 8). Initially the program should display the board as a 5×5 grid of asterisks. The user then enters the coordinates of a cell. The program should display the grid with the character at those coordinates now displayed. The user then enters coordinates of another cell. The program should now display the grid with the previous character and the new character displayed. If the two characters match, then they should permanently replace the asterisks in those locations. Otherwise, when the user enters the next set of coordinates, those characters should be replaced by asterisks. The game continues this way until the player matches

everything or runs out of turns. You can decide how many turns they player gets.

Ask the user to enter the numerator and denominator of a fraction, and the digit they want to know. For instance, if the user enters a numerator of 1 and a denominator of 7 and wants to know the 4th digit, your program should print out

8, because

$\frac{1}{7} = .142856...$ and 8 is the 4th digit. One way to do this is to mimic the long division process you may have learned in grade school. It can be done in about five lines using the // operator at one point in the program.

Randomly generate a 6×6 list that has exactly 12 ones placed in random locations in the list. The rest of the entries should be zeroes.

Randomly generate a 9×9 list where the entries are integers between 1 and 9 with no repeat entries in any row or in any column.

Miscellaneous Topics II

In this chapter we will look at variety of useful things to know.

10.1 str, int, float, and list

The **str**, **int**, **float**, and **list** functions are used to convert one data type into another.

str Quite often we will want to convert a number to a string to take advantage of string methods to break the number apart. The built-in function str is used to convert things into strings. Here are some examples:

Statement	Result
str (37)	'37'
str (3.14)	'3.14'
str ([1,2,3])	'[1,2,3]'

int and float The **int** function converts something into an integer. The **float** function converts something into a floating point number. Here are some examples.

```
int('37') 37
float('3.14') 3.14
int(3.14) 3
```

To convert a float to an integer, the **int** function drops everything after the decimal point.

87

list The list function takes something that can be converted into a list and makes into a list. Here are two uses of it.

```
list(range(5)) [0,1,2,3,4]
list('abc') ['a','b','c']
```

Examples

Example 1 Here is an example that finds all the palindromic numbers between 1 and 10000. A palindromic number is one that is the same backwards as forwards, like 1221 or 64546.

```
for i in range(1,10001):
    s = str(i) if s==s[::-1]: print(s)
```

We use the **str** function here to turn the integer i into a string so we can use slices to reverse it.

Example 2 Here is an example that tells a person born on January 1, 1991 how old they are in

2010.

```
birthday = 'January 1, 1991' year = int(birthday[-4:])
print('You are', 2010-year, 'years old.')
```

The year is in the last four characters of birthday. We use **int** to convert those characters into an integer so we can do math with the year.

Example 3 Write a program that takes a number num and adds its digits. For instance, given the number 47, the program should return 11 (which is 4+7). Let us start with a 2-digit example.

```
digit = str(num)
answer = int(digit[0]) + int(digit[1])
```

The idea here is that we convert num to a string so that we can use indexing to get the two digits separately. We then convert each back to an integer using the int function. Here is a version that handles numbers with arbitrarily many digits:

```
digit = str(num) answer = 0
for i in range(len(digit)):
    answer = answer + int(digit[i])
```

We can do the above program in a single line using a list comprehension.

```
answer = sum([int(c) for c in str(num)])
```

10.2. BOOLEANS

Example 4 To break a decimal number, num, up into its integer and fractional parts, we can do the following:

```
ipart = int(num) dpart = num - int(num)
```

For example, if num is 12.345, then ipart is 12 and dpart is $12.345 - 12 = .345$.

Example 5 If we want to check to see if a number is prime, we can do so by checking to see if it has any divisors other than itself and 1. In Section 9.4 we saw code for this, and we had the following for loop:

```
for i in range(2,num):
```

This checks for divisibility by the integers 2, 3, ..., num-1. However, it turns out that you really only have to check the integers from 2 to the square root of the number. For instance, to check if 111 is prime, you only need to check if it is divisible by the integers 2 through 10, as $p^{111} \approx 10.5$. We could then try the following for loop:

```
for i in range(2,num**.5):
```

However, this gives an error, because num**.5 might not be an integer, and the range function needs integers. We can use int to correct this:

```
for i in range(2,int(num**.5)+1):
```

The +1 at the end is needed due to the range function not including the last value.

10.2 Booleans

Boolean variables in Python are variables that can take on two values, **True** and **False**. Here are two examples of setting Boolean variables:

```
game_over = True highlight_text = False
```


Booleans can help make your programs more readable. They are often used as flag variables or to indicate options. Booleans are often used as conditions in if statements and while loops:

```
if game_over:  
    print('Bye!')
```

Note the following equivalences:

```
if game_over: ⇔ if game_over==True: while not game_over: ⇔ while  
game_over==False:
```

note Conditional expressions evaluate to booleans and you can even assign them to variables. For instance, the following assigns **True** to x because 6==6 evaluates to **True**.

```
x = (6==6)
```

We have seen booleans before. The isalpha string method returns **True** if every character of the string is a letter and **False** otherwise.

10.3 Shortcuts

Shortcut operators Operations like count=count+1 occur so often that there is a shorthand for them. Here are a couple of examples:

Statement Shorthand

```
count=count+1 count+=1
```

```
total=total-5 total-=5
```

```
prod=prod*2 prod*=2
```

There are also shortcut operators /=, %=, //=, and **=.

An assignment shortcut Look at the code below.

```
a = 0 b = 0 c = 0
```

A nice shortcut is:

```
a = b = c = 0
```

- Another assignment shortcut

Say we have a list L with three elements in it, and we want to assign those elements to variable names. We could do the following:

```
x = L[0] y = L[1] z = L[2]
```

Instead, we can do this:

```
x,y,z = L
```

Similarly, we can assign three variables at a time like below:

```
x,y,z = 1,2,3
```

And, as we have seen once before, we can swap variables using this kind of assignment.

```
x,y,z = y,z,x
```

- Shortcuts with conditions

Here are some handy shortcuts:

Statement Shortcut

```
ifa==0andb==0andc==0: ifa==b==c==0:
```

```
if 1<aanda<bandb<5: if1<a<b<5:
```

10.4. *SHORT-CIRCUITING*

10.4 Short-circuiting

Say we are writing a program that searches a list of words for those whose fifth character is 'z'. We might try the following:

```
for w in words:
```

```
if w[4]=='z': print(w)
```

But with this, we will occasionally get a string index out of range error. The problem is that some words in the list might be less than five characters long. The following if statement, however, will work:

```
if len(w)>=5 and w[4]=='z':
```

It might seem like we would still get an error because we are still checking `w[4]`, but there is no error. The key to why this works is *short-circuiting*. Python starts by checking the first part of the condition, `len(w)>=5`. If that condition turns out to be false, then the whole **and** condition is guaranteed to be false, and so there is no point in even looking at the second condition. So Python doesn't bother with the second condition. You can rely on this behavior.

Short-circuiting also happens with **or** conditions. In this case, Python checks the first part of the **or** and if it is true, then the whole **or** is guaranteed to be true, and so Python will not bother checking the second part of the **or**.

10.5 Continuation

Sometimes you'll write a long line of code that would be more readable if it were split across two lines. To do this, use a backslash `\` character at the end of the line to indicate that the statement continues onto the next line. Here is an example:

```
if 'a' in string or 'b' in string or 'c' in string \ or 'd' in string or 'e' in string:
```

Make sure there are no extra spaces after the backslash or you will get an error message.

If you are entering a list, dictionary, or the arguments of a function, the backslash can be left out:

```
L = [ 'Joe', 'Bob', 'Sue', 'Jimmy', 'Todd', 'Frank',  
      'Mike', 'John', 'Amy', 'Edgar', 'Sam']
```

10.6 pass

The **pass** statement does nothing. Believe it or not, such a thing does have a few uses that we will see later.

10.7 String formatting

Suppose we are writing a program that calculates a 25% tip on a bill of \$23.60. When we multiply, we get 5.9, but we would like to display the result as \$5.90, not \$5.9. Here is how to do it:

```
a = 23.60 * .25
```

```
print('The tip is {:.2f}'.format(a))
```

This uses the **format** method of strings. Here is another example:

```
bill = 23.60  
tip = 23.60 * .25
```

```
print('Tip: ${:.2f}, Total: ${:.2f}'.format(tip, bill+tip))
```

The way the **format** method works is we put a pair of curly braces `{}` anywhere that we want a formatted value. The arguments to the **format** function are the values we want formatted, with the first argument matching

up with the first set of braces, the second argument with the second set of braces, etc. Inside each set of curly braces you can specify a formatting code to determine how the corresponding argument will be formatted.

Formatting integers To format integers, the formatting code is `{:d}`. Putting a number in front of the `d` allows us to right-justify integers. Here is an example: `print('{:3d}'.format(2)) print('{:3d}'.format(25)) print('{:3d}'.format(138))`

```
2
25
138
```

The number 3 in these examples says that the value is allotted three spots. The value is placed as far right in those three spots as possible and the rest of the slots will be filled by spaces. This sort of thing is useful for nicely formatting tables.

To center integers instead of right-justifying, use the `^` character, and to leftjustify, use the `<` character.

```
print('{:^5d}'.format(2)) print('{:^5d}'.format(222))
print('{:^5d}'.format(13834))
2
122
13834
```

Each of these allots five spaces for the integer and centers it within those five spaces.

Putting a comma into the formatting code will format the integer with commas. The example below prints 1,000,000:

```
print('{:,d}'.format(1000000))
10.8. NESTED LOOPS
```

Formatting floats To format a floating point number, the formatting code is `{:f}`. To only display the number to two decimal places, use `{:.2f}`. The 2 can be changed to change the number of decimal places.

You can right-justify floats. For example, `{:8.2f}` will allot eight spots for its value—one of those is for the decimal point and two are for the part of the value after the decimal point. If the value is 6.42, then only four spots are needed and the remaining spots are filled by spaces, causing the value to be right-justified.

The `^` and `<` characters center and left-justify floats.

Formatting strings To format strings, the formatting code is `{:s}`. Here is an example that centers some text:

```
print('{:^10s}'.format('Hi')) print('{:^10s}'.format('there!'))
```

Hi there!

To right-justify a string, use the `>` character:

```
print('{:>6s}'.format('Hi')) print('{:>6s}'.format('There'))
```

Hi there!

There is a whole lot more that can be done with formatting. See the Python documentation [1].

10.8 Nested loops

You can put loops inside of other loops. A loop inside of another loop is said to be *nested*, and you can, more or less, nest loops as deeply as you want.

Example 1 Print a 10×10 multiplication table.

```
for i in range(1,11):
    for j in range(1,11):
        print('{:3d}'.format(i*j), end=' ')
    print()
```

A multiplication table is a two-dimensional object. To work with it, we use two for loops, one for the horizontal direction and one for the vertical direction. The print statement right justifies the products to make them look nice. The `end=""` allows us to print several things on each row. When we are done printing a row, we use `print()` to advance things to the next line.

Example 2 A common math problem is to find the solutions to a system of equations. Sometimes you want to find only the integer solutions, and this can be a little tricky mathematically. However, we can write a program that does a brute force search for solutions. Here we find all the integer solutions

(x, y) to the system $2x + 3y = 4$, $x - y = 7$, where x and y are both between -50 and 50.

```
for x in range(-50,51):
for y in range(-50,51):
if 2*x+3*y==4 and x-y==7:
print(x,y)
```

Example 3 A Pythagorean triple is a triple of numbers (x, y,z) such that $x^2 + y^2 = z^2$. For instance (3,4,5) is a Pythagorean triple because $3^2 + 4^2 = 5^2$. Pythagorean . Pythagorean triangle). Here is a program that finds all the Pythagorean triples (x, y,z) where x, y, and z are positive and less than 100.

```
for x in range(1,100):
for y in range(1,100):
for z in range(1,100):
if x**2+y**2==z**2: print(x,y,z)
```

If you run the program, you'll notice that there are redundant solutions. For instance, (3,4,5) and (4,3,5) are both listed. To get rid of these redundancies, change the second loop so that it runs from x to 100. This way, when x is 4, for instance, the first value for y that will be searched is 4, rather than 1, and so we won't get the redundant (4,3,5). Also change the third loop so that it runs from y to 100.

As you look through the solutions, you might also notice that there are many solutions that are multiples of others, like (6,8,10), and (9,12,15) are multiples of (3,4,5). The following program finds only primitive Pythagorean triples, those that aren't multiples of another triple. The way it does this is every time a new triple is found, it checks to make sure that x, y, and z are not all divisible by the same number.

```
for x in range(1,100):
for y in range(x,100):
for z in range(y,100):
if x**2+y**2==z**2:
for i in range(2,x):
if x%i==0 and y%i==0 and z%i==0: break
```

else:

print((x,y,z), end=' ')

Example 4 In Section 15.7, we will write a game to play tic-tac-toe. The board is a 3×3 grid, and we will use nested for loops to create it.

10.9. EXERCISES

Example 5 Your computer screen is grid of pixels. To draw images to the screen, we often use nested for loops—one loop for the horizontal direction, and one for the vertical direction. See Sections 18.2 and 22.6 for examples.

Example 6 List comprehensions can contain nested for loops. The example below returns a list of all the vowels in a list of words.

```
[char for item in L for char in item if char in 'aeiou']
```

10.9 Exercises

Write a program that uses **list** and **range** to create the list [3,6,9, ...,99].

Write a program that asks the user for a weight in kilograms. The program should convert the weight to kilograms, formatting the result to one decimal place.

Write a program that asks the user to enter a word. Rearrange all the letters of the word in alphabetical order and print out the resulting word. For example, *abracadabra* should become *aaaaabbcdrr*.

Write a program that takes a list of ten prices and ten products, applies an 11% discount to each of the prices displays the output like below, right-justified and nicely formatted.

Apples \$ 2.45

Oranges \$ 18.02

...

Pears \$120.03

Use the following two lists and the **format** method to create a list of card names in the format *card value of suit name* (for example, 'Two of Clubs').

suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades'] values = ['One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace']

Write a program that uses a boolean flag variable in determining whether two lists have any items in common.

Write a program that creates the list [1,11,111,1111,...,111...1], where the entries have an ever increasing number of ones, with the last entry having 100 ones. Write a program to find all numbers between 1 and 1000 that are divisible by 7 and end in a

6.

Write a program to determine how many of the numbers between 1 and 10000 contain the digit 3.

Adding certain numbers to their reversals sometimes produces a palindromic number. For instance, $241 + 142 = 383$. Sometimes, we have to repeat the process. For instance, $84 + 48 = 132$ and $132 + 231 = 363$. Write a program that finds both two-digit numbers for which this process must be repeated more than 20 times to obtain a palindromic number.

Write a program that finds all pairs of six-digit palindromic numbers that are less than 20 apart. One such pair is 199991 and 200002.

The number 1961 reads the same upside-down as right-side up. Print out all the numbers between 1 and 100000 that read the same upside-down as right-side up.

The number 99 has the property that if we multiply its digits together and then add the sum of its digits to that, we get back to 99. That is, $(9 \times 9) + (9 + 9) = 99$. Write a program to find all of the numbers less than 10000 with this property. (There are only nine of them.)

Write a program to find the smallest positive integer that satisfies the following property: If you take the leftmost digit and move it all the way to the right, the number thus obtained is exactly 3.5 times larger than the original number. For instance, if we start with 2958 and move the 2 all the way to the right, we get 9582, which is roughly 3.2 times the original number.

Write a program to determine how many zeroes 1000! ends with.

Write a program that converts a decimal height in feet into feet and inches.

For instance, an

input of 4.75 feet should become 4feet,9inches.

Write a program that repeatedly asks the user to enter a height in the format *feet'inches"* (like 5'11" or 6'3". The user indicates they are done entering heights (like 5'11" or 6'3". The user indicates they are done entering heights footers, 6-footers, and 7-footers were entered.

Write a program that repeatedly asks the user to enter a football score in the format *winning score-losing score* (like 27-13 or 21-3). The user indicates they are done entering scores by entering done. The program should then output the highest score and the lowest score out of all the scores entered.

Write a program that repeatedly asks the user to enter a birthday in the format *month/day*

(like 12/25 or 2/14). The user indicates they are done entering birthdays by entering done. The program should return a count of how many of those birthdays are in February and how many are on the 25th of some month (any month).

Write a program that asks the user to enter a date in the format mm/dd/yy and converts it to a more verbose format. For example, 02/04/77 should get converted into February4, 1977.

Write a program that asks the user to enter a fraction in the form of a string like '1/2' or '8/24'. The program should reduce the fraction to lowest terms and print out the result.

Write a program to find all four solutions to the following problem: If a starfruit is worth \$5, a mango is worth \$3, and three oranges together cost \$1, how many starfruits, mangoes, and oranges, totaling 100, can be bought for \$100?

10.9. EXERCISES

The currency of a strange country has coins worth 7 cents and 11 cents. Write a program to determine the largest purchase price that cannot be paid using these two coins.

Here is an old puzzle you can solve using brute force by using a computer program to check all the possibilities: In the calculation $43 + 57 = 207$, every digit is precisely one away from its true value. What is the correct calculation?

Write a program that finds all integer solutions to Pell's equation $x^2 - 2y^2 = 1$, where x and y are between 1 and 100.

Write a program that asks the user for a number and prints out all the ways to write the number as difference of two perfect squares, $x^2 - y^2$, where x and y are both between 1 and 1000. Writing a number as a difference of two squares leads to clever techniques for factoring large numbers.

Write a program that simulates all possible rolls of four dice and for each simulated roll, finds the sums of pairs of dice. For instance, if the roll is 5 1 2 4, the sums are 6, 8, 9, 3, 5, and 6. For each of the possible sums from 2 to 12, find the total number of simulated rolls in which the sum appears and what percentage of the simulated rolls had those sums appear. Output the totals and percentages, nicely formatted, with the percentages formatted to one decimal place. To check your work, you should find that the sum 2 comes up in 171 rolls, which is 13.2% of the rolls.

In a magic square, each row, each column, and both diagonals add up to the same number. A partially filled magic square is shown below. Write a program to check through all the possibilities to fill in the magic square.

```
5 _ _  
_ 6 2  
3 8 _
```

The following is useful as part of a program to play *Minesweeper*. Suppose you have a 5×5 list that consists of zeros and M 's. Write a program that creates a new 5×5 list that has M 's in the same place, but the zeroes are replaced by counts of how many M 's are in adjacent cells (adjacent either

horizontally, vertically, or diagonally). An example is shown below. [Hint: short-circuiting may be helpful for avoiding index-out-of-range errors.]

```
0 M 0 M 0 1 M 3 M
1
```

```
0 0 M 0 0 1 2 M 2 1
0 0 0 0 0 2 3 2 1 0
```

```
M M 0 0 0 M M 2 1
1
0 0 0 M 0 2 2 2 M 1
```

Pascal's triangle is shown below. On the outside are 1's and each other number is the sum of the two numbers directly above it. Write a program to generate Pascal's triangle. Allow the user to specify the number of rows. Be sure that it is nicely formatted, like below.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Given two dates entered as strings in the form mm/dd/yyyy where the years are between 1901 and 2099, determine how many days apart they are. Here is a bit of information that may be useful: Leap years between 1901 and 2099 occur exactly every four years, starting at 1904. February has 28 days, 29 during a leap year. November, April, June, and September each have 30 days. The other months have 31 days.

Monte Carlo simulations can be used to estimate all sorts of things, including probabilities of coin flip and dice events. As an example, to estimate the probability of rolling a pair of sixes with two dice, we could use random integers to simulate the dice and run the simulation thousands of times, counting what percentage of the time a pair of sixes comes up.

Estimate the probability of rolling a *Yahtzee* in a single roll of five dice. That is estimate the probability that when rolling five dice they all come out to be the same number.

Estimate the probability of rolling a large straight in a single roll of five dice. A large straight is a roll where the dice come out 1-2-3-4-5 or 2-3-4-5-6 in any order.

Estimate the average longest run of heads or tails when flipping a coin 200 times. Estimate the average number of coin flips it takes before five heads in a row come up.

Estimate the average number of coin flips it takes before the string *s* comes up, where *s* is a string of heads and tails, like HHTTH.

Dictionaries

A dictionary is a more general version of a list. Here is a list that contains the number of days in the months of the year:

```
days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

If we want the number of days in January, use `days[0]`. December is `days[11]` or `days[-1]`.

Here is a dictionary of the days in the months of the year:

```
days = {'January':31, 'February':28, 'March':31, 'April':30,  
'May':31, 'June':30, 'July':31, 'August':31,  
'September':30, 'October':31, 'November':30, 'December':31}
```

To get the number of days in January, we use `days['January']`. One benefit of using dictionaries here is the code is more readable, and we don't have to figure out which index in the list a given month is at. Dictionaries have a number of other uses, as well.

11.1 Basics

Creating dictionaries Here is a simple dictionary:

```
d = {'A':100, 'B':200}
```

To declare a dictionary we enclose it in curly braces, `{}`. Each entry consists of a pair separated by a colon. The first part of the pair is called the *key* and the second is the *value*. The key acts like an index. So in the first pair, `'A':100`, the key is `'A'`, the value is 100, and `d['A']` gives 100. Keys are often

strings, but they can be integers, floats, and many other things as well. You can mix different types of keys in the same dictionary and different types of values, too.

Changing dictionaries Let's start with this dictionary:

```
d = {'A':100, 'B':200}
```

99

To change d['A'] to 400, do

```
d['A']=400
```

To add a new entry to the dictionary, we can just assign it, like below:

```
d['C']=500
```

Note that this sort of thing does not work with lists. Doing L[2]=500 on a list with two elements would produce an index out of range error. But it does work with dictionaries.

To delete an entry from a dictionary, use the **del** operator:

```
del d['A']
```

Empty dictionary The empty dictionary is {}, which is the dictionary equivalent of [] for lists or "" for strings.

Important note The order of items in a dictionary will not necessarily be the order in which put them into the dictionary. Internally, Python rearranges things in a dictionary in order to optimize performance.

11.2 Dictionary examples

Example 1 You can use a dictionary as an actual dictionary of definitions: d

```
= {'dog' : 'has a tail and goes woof!',
```

```
'cat' : 'says meow',
```

```
'mouse' : 'chased by cats'}
```

Here is an example of the dictionary in use:

```
word = input('Enter a word: ') print('The definition is:', d[word])
```

Enter a word: mouse

The definition is: chased by cats

Example 2 The following dictionary is useful in a program that works with Roman numerals. `numerals = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}` **Example 3** In the game Scrabble, each letter has a point value associated with it. We can use the following dictionary for the letter values: `points = {'A':1, 'B':3, 'C':3, 'D':2, 'E':1, 'F':4, 'G':2, 'H':4, 'I':1, 'J':8, 'K':5, 'L':1, 'M':3, 'N':1,`

`'O':1, 'P':3, 'Q':10, 'R':1, 'S':1, 'T':1, 'U':1, 'V':4, 'W':4, 'X':8, 'Y':4, 'Z':10}`

11.3. WORKING WITH DICTIONARIES

To score a word, we can do the following:

```
score = sum([points[c] for c in word])
```

Or, if you prefer the long way:

```
total = 0 for c in word:
```

```
total += points[c]
```

Example 4 A dictionary provides a nice way to represent a deck of cards:

```
deck = [{'value':i, 'suit':c} for c in ['spades', 'clubs', 'hearts', 'diamonds'] for i in range(2,15)]
```

The deck is actually a list of 52 dictionaries. The `shuffle` method can be used to shuffle the deck:

```
shuffle(deck)
```

The first card in the deck is `deck[0]`. To get the value and the suit of the card, we would use the following:

```
deck[0]['value'] deck[0]['suit']
```

11.3 Working with dictionaries

Copying dictionaries Just like for lists, making copies of dictionaries is a little tricky for reasons we will cover later. To copy a dictionary, use its `copy` method. Here is an example:

```
d2 = d.copy()
```

in The `in` operator is used to tell if something is a key in the dictionary. For instance, say we have the following dictionary:

```
d = {'A':100, 'B':200}
```

Referring to a key that is not in the dictionary will produce an error. For

instance, `print(d['C'])` will fail. To prevent this error, we can use the `in` operator to check first if a key is in the dictionary before trying to use the key. Here is an example:

```
letter = input('Enter a letter: ') if letter in d:
    print('The value is', d[letter])
else:
    print('Not in dictionary')
```

You can also use `notin` to see if a key is not in the dictionary.

Looping Looping through dictionaries is similar to looping through lists. Here is an example that prints the keys in a dictionary:

```
for key in d:
    print(key)
```

Here is an example that prints the values:

```
for key in d:
    print(d[key])
```

Lists of keys and values The following table illustrates the ways to get lists of keys and values from a dictionary. It uses the dictionary `d={'A':1,'B':3}`.

Statement	Result	Description
<code>list(d)</code>	<code>['A','B']</code>	keys of d
<code>list(d.values())</code>	<code>[1,3]</code>	values of d
<code>list(d.items())</code>	<code>[('A',1),('B',3)]</code>	(key,value) pairs of d

The pairs returned by `d.items` are called *tuples*. Tuples are a lot like lists.

They are covered in Section [19.2](#).

Here is a use of `d.items` to find all the keys in a dictionary `d` that correspond to a value of 100:

```
d = {'A':100, 'B':200, 'C':100}
L = [x[0] for x in d.items() if x[1]==100]
['A', 'C']
```

dict The `dict` function is another way to create a dictionary. One use for it is kind of like the opposite of the `items` method:

```
d = dict([('A',100),('B',300)])
```

This creates the dictionary `{'A':100,'B':300}`. This way of building a

dictionary is useful if your program needs to construct a dictionary while it is running.

Dictionary comprehensions Dictionary comprehensions work similarly to list comprehensions. The following simple example creates a dictionary from a list of words, where the values are the lengths of the words:

```
d = {s : len(s) for s in words}
```

11.4 Counting words

We can use dictionaries to count how frequently certain words appear in a text. *11.4. COUNTING WORDS*

In Section 12.1, we will learn how to read from a text file. For now, here's a line of code that reads the entire contents of a file containing the text of Shakespeare's *Romeo and Juliet* and stores the contents in a string called `text`:

```
text = open('romeoandjuliet.txt').read()
```

To get at the individual words, we will use the `split` method to turn the string into a list of its individual words. Also, because some words may be capitalized, we will convert the whole string to lowercase. We also have to remove punctuation.

```
from string import punctuation
text = text.lower()
for p in punctuation:
    text = text.replace(p, "")
words = text.split()
```

Next comes the dictionary code that does the counting. The dictionary keys will be the words from the text and the values will be counts of how many time each word appears. We start with an empty dictionary. Then for every word in the list of words, if we have seen the word before, we add one to its count, and otherwise we set the count for that word equal to 1. Here is the code:

```
d = {}
for w in words:
    if w in d: d[w] = d[w] + 1
    else:
```


`d[w] = 1`

Once we have created the dictionary, we can use the following code to print the items in alphabetical order:

```
items = list(d.items()) items.sort() for i in items:
print(i)
```

The way this works is a little tricky. Remember that `d.items()` returns a list of pairs (called tuples), which are a lot like lists. When we sort a list of tuples, the sorting is done by the first entry, which in this case is the word. So the sorting is done alphabetically.

If we instead want to order things by frequency, we can flip the order of the tuples and then sort:

```
items = list(d.items()) items = [(i[1], i[0]) for i in items] items.sort() for i in
items:
print(i)
```

Here is the code all together:

```
from string import punctuation
```

```
# read from file, remove caps and punctuation, and split into words text =
open('romeoandjuliet.txt').read()
```

```
text = text.lower() for p in punctuation: text = text.replace(p, "") words =
text.split()
```

```
# build the dictionary of frequencies d = {} for w in words:
```

```
if w in d: d[w] = d[w] + 1 else:
```

```
d[w] = 1
```

```
# print in alphabetical order items = list(d.items()) items.sort() for i in items:
print(i)
```

```
# print in order from least to most common items = list(d.items()) items =
[(i[1], i[0]) for i in items] items.sort() for i in items:
print(i)
```

See Section [24.5](#) for another approach to word frequencies.

11.5 Exercises

Write a program that repeatedly asks the user to enter product names and prices. Store all of these in a dictionary whose keys are the product names and whose values are the prices. When the user is done entering products and prices, allow them to repeatedly enter a product name and print the corresponding price or a message if the product is not in the dictionary.

Using the dictionary created in the previous problem, allow the user to enter a dollar amount and print out all the products whose price is less than that amount.

For this problem, use the dictionary from the beginning of this chapter whose keys are month names and whose values are the number of days in the corresponding months.

Ask the user to enter a month name and use the dictionary to tell them how many days are in the month.

Print out all of the keys in alphabetical order.

Print out all of the months with 31 days.

Print out the (key-value) pairs sorted by the number of days in each month

11.5. EXERCISES

(e) Modify the program from part (a) and the dictionary so that the user does not have to know how to spell the month name exactly. That is, all they have to do is spell the first three letters of the month name correctly.

Write a program that uses a dictionary that contains ten user names and passwords. The program should ask the user to enter their username and password. If the username is not in the dictionary, the program should indicate that the person is not a valid user of the system. If the username is in the dictionary, but the user does not enter the right password, the program should say that the password is invalid. If the password is correct, then the program should tell the user that they are now logged in to the system.

Repeatedly ask the user to enter a team name and the how many games the team won and how many they lost. Store this information in a dictionary where the keys are the team names and the values are lists of the form [*wins*, *losses*].

Using the dictionary created above, allow the user to enter a team name and print out the team's winning percentage.

Using the dictionary, create a list whose entries are the number of wins of each team.

Using the dictionary, create a list of all those teams that have winning records.

Repeatedly ask the user to enter game scores in a format like *team1 score1 team2 score2*. Store this information in a dictionary where the keys are the team names and the values are lists of the form *[wins, losses]*.

Create a 5×5 list of numbers. Then write a program that creates a dictionary whose keys are the numbers and whose values are the how many times the number occurs. Then print the three most common numbers.

Using the card dictionary from earlier in this chapter, create a simple card game that deals two players three cards each. The player with the highest card wins. If there is a tie, then compare the second highest card and, if necessary, the third highest. If all three cards have the same value, then the game is a draw.

Using the card dictionary from earlier in the chapter, deal out three cards. Determine the following:

If the three cards form a flush (all of the same suit)

If there is a three-of-a-kind (all of the same value)

If there is a pair, but not three-of-a-kind

If the three cards form a straight (all in a row, like (2, 3, 4) or (10, Jack, Queen))

Using the card dictionary from earlier in the chapter run a Monte Carlo simulation to estimate the probability of being dealt a flush in a five card hand. See Exercise 32 of Chapter 10 for more about Monte Carlo simulations.

In Section 6.10 we met the substitution cipher. This cipher replaces every letter with a different letter. For instance every *a* might be replaced with an *e*, every *b* might be replaced with an

a , etc. Write a program that asks the user to enter two strings. Then determine if the second string could be an encoded version of the first one with a substitution cipher. For instance, CXYZ is not an encoded version of BOOK because O got mapped to two separate letters. Also, CXXK is not an encoded version of BOOK, because K got mapped to itself. On the other hand, CXXZ would be an encoding of BOOK. This problem can be done with or without a dictionary.

Below are the notes used in music:

C C# D D# E F F# G G# A A# B

The notes for the C major chord are C, E, G. A mathematical way to get this is that E is 4 steps past C and G is 7 steps past C. This works for any base. For example, the notes for D major are D, F#, A. We can represent the major chord steps as a list with two elements: [4,7]. The corresponding lists for some other chord types are shown below:

Minor [3,7] Augmented fifth [4,8] Minor fifth [4,6] Major sixth [4,7,9]
Dominant seventh [4,7,10] Minor seventh [3,7,10] Major seventh [4,7,11]

Diminished [3,6,10] seventh
Minor sixth [3,7,9]

Write a program that asks the user for the key and the chord type and prints out the notes of the chord. Use a dictionary whose keys are the (musical) keys and whose values are the lists of steps.

Suppose you are given the following list of strings:

L = ['aabaabac', 'cabaabca', 'aaabbcba', 'aabacbab', 'acababba']

Patterns like this show up in many places, including DNA sequencing. The user has a string of their own with only some letters filled in and the rest as asterisks. An example is a**a****. The user would like to know which of the strings in the list fit with their pattern. In the example just given, the matching strings are the first and fourth. One way to solve this problem is to create a dictionary whose keys are the indices in the user's string of the non-asterisk characters and whose values are those characters. Write a program implementing this approach (or some other approach) to find the strings that

match a user-entered string.

Dictionaries provide a convenient way to store structured data. Here is an example dictionary:

```
d=[ {'name':'Todd', 'phone':'555-1414', 'email':'todd@mail.net'},  
    {'name':'Helga', 'phone':'555-1618', 'email':'helga@mail.net'},  
    {'name':'Princess', 'phone':'555-3141', 'email':''},  
    {'name':'LJ', 'phone':'555-2718', 'email':'lj@mail.net'}]
```

Write a program that reads through any dictionary like this and prints the following:

All the users whose phone number ends in an 8

All the users that don't have an email address listed

11.5. EXERCISES

The following problem is from Chapter 6. Try it again, this time using a dictionary whose keys are the names of the time zones and whose values are offsets from the Eastern time zone.

Write a program that converts a time from one time zone to another. The user enters the time in the usual American way, such as 3:48pm or 11:26am. The first time zone the user enters is that of the original time and the second is the desired time zone. The possible time zones are Eastern, Central, Mountain, or Pacific.

Time: 11:48pm

Starting zone: Pacific

Ending zone: Eastern

2:48am

(a) Write a program that converts Roman numerals into ordinary numbers. Here are the conversions: M=1000, D=500, C=100, L=50, X=10, V=5 I=1. Don't forget about things like IV being 4 and XL being 40.

(b) Write a program that converts ordinary numbers into Roman numerals

Text Files

There is a ton of interesting data to be found on the internet stored in text files. In this chapter we will learn how to work with data stored in text files.

12.1 Reading from files

Suppose we have a text file called `example.txt` whose contents are shown below, and we want to read its contents into Python. There are several ways to do so. We will look at two of them.

Hello.

This is a text file.

Bye!

The first way to read a text file uses a list comprehension to load the file line-byline into a list:

```
lines = [line.strip() for line in open('example.txt')]
```

The list `lines` is now

```
['Hello.', 'This is a text file.', 'Bye!']
```

The string method `strip` removes any whitespace characters from the beginning and end of a string. If we had not used it, each line would contain a newline character at the end of the line. This is usually not what we want.

Note: `strip` removes whitespace from both the beginning and end of the line. Use `rstrip` if you need to preserve whitespace at the beginning of the line.

The second way of reading a text file loads the entire file into a string: `s =`

```
open('example.txt').read()
```

The string `s` is now

```
'Hello.\nThis is a text file.\nBye!'
```

109

Directories

Say your program opens a file, like below:

```
s = open('file.txt').read()
```

The file is assumed to be in the same directory as your program itself. If it is in a different directory, then you need to specify that, like below:

```
s = open('c:/users/heinold/desktop/file.txt').read()
```

12.2 Writing to files

There are also several ways to write to files. We will look at one way here. We will be writing to a file called writefile.txt.

```
f = open('writefile.txt', 'w') print("This is line 1.", file=f) print("This is line 2.", file=f) f.close()
```

We first have to open the file. That is what the first line does, with the 'w' indicating that we want to be able to write to the file. Python creates what is called a file object to represent the file, and we give that object the name f. This is what we use to refer to the file. To write to the file, we use the print statement with the optional file argument that specifies the file to write to. When we are done writing, we should close the file to make sure all of our changes take. Be careful here because if writefile.txt already exists, its contents will be overwritten.

12.3 Examples

Example 1 Write a program that reads a list of temperatures from a file called temps.txt, converts those temperatures to Fahrenheit, and writes the results to a file called ftemps.txt.

```
file1 = open('ftemps.txt', 'w') temperatures = [line.strip() for line in open('temps.txt')] for t in temperatures: print(t*9/5+32, file=file1) file1.close()
```

Example 2 In Section 7.6 we wrote a simple quiz game. The questions and answers were both contained in lists hard-coded into the program. Instead of that, we can store the questions and answers in files. That way, if you decide to change the questions or answers, you just have to change their files. Moreover, if you decide to give the program to someone else who doesn't know

120 12.4. WORDPLAY

Python, they can easily create their own lists of questions and answers. To do this, we just replace the lines that create the lists with the following: questions = [line.strip() for line in open('questions.txt')] answers = [line.strip() for line

```
in open('answers.txt')]
```

Example 3 Say you have a text file that contains the results of every 2009-10 NCAA basketball game. (You can find such a file at www.kenpom.com.) A typical line of the file looks like this:

02/27/2010, Robert Morris, 61, Mount St. Mary's, 63

Below is a program that scans through the file to find the most lopsided game, the one where the winning team had the largest margin of victory.

```
lines = [line.strip() for line in open('scores.txt')] games = [line.split(',') for line in lines] print(max([abs(int(g[2])-int(g[4])) for g in games]))
```

We use the split method to break each line into a lists of its component parts. The scores are at indices 2 and 4. To find the maximum difference, we can use a list comprehension to create a list of all the margins of victories and use **max** to find the maximum.

The maximum turns out to be 84. Unfortunately, the method above does not tell us anything else about the game. In order to do that, we resort to the longer way to find maximums, described in Section 5.5. This allows us to store information about the game as we search for the largest margin of victory.

```
lines = [line.strip() for line in open('scores.txt')] games = [line.split(',') for line in lines]
biggest_diff = 0
for g in games:
    diff = abs(int(g[2])-int(g[4]))
    if diff>biggest_diff: biggest_diff = diff
    game_info = g
```

```
print (game_info)
['12/03/2009', ' SalemInternational', '35', ' Marshall', '119']
```

12.4 Wordplay

If you like words, you can have a lot of fun with a wordlist, which is a text file where each line contains a different word. A quick web search will turn up a variety of different wordlists, ranging from lists of common English words to lists containing practically every English word.

Assuming the wordlist file is wordlist.txt, we can load the words into a list using the line below.

```
wordlist = [line.strip() for line in open('wordlist.txt')]
```

Example 1 Print all three letter words.

```
for word in wordlist:
```

```
if len(word)==3: print(word)
```

Note that this and most of the upcoming examples can be done with list comprehensions:

```
print([word for word in wordlist if len(word)==3])
```

Example 2 Print all the words that start with gn or kn.

```
for word in wordlist:
```

```
if word[:2]=='gn' or word[:2]=='kn':
```

```
print(word)
```

Example 3 Determine what percentage of words start with a vowel.

```
count = 0 for word in wordlist:
```

```
if word[0] in 'aeiou': count=count+1
```

```
print(100*count/len(wordlist))
```

Example 4 Print all 7-letter words that start with th and end in ly. Things like this are good for cheating at crosswords.

```
for word in wordlist:
```

```
if len(word)==7 and word[:2]=='th' and word[-2:]=='ly':
```

```
print(word)
```

Example 5 Print the first ten words that start with q.

```
i=0 while wordlist[i][0]!='q': i=i+1 print(wordlist[i:i+10])
```

Note this is not a very efficient way of doing things since we have to scan through most of the list.

A binary search would be more efficient, but the above approach still runs almost instantly even for large files.

Example 6 Find the longest word that can be made using only the letters a, b, c, d, and e.

```
largest = 0 for word in wordlist:
```

```
for c in word:
    if c not in 'abcde':
        break
    else:
        if len(word)>largest: largest=len(word) largest_word=word
print(largest_word)
```

The way this program works is for every word in the wordlist, we use a for/else loop (Section 9.4) to scan through the word looking checking each character to see if it is an a, b, c, d, or e. If any letter isn't one of these, then we break out of the loop and move on to the next word. On the other hand, if we get all the way through the loop, then we go to else block. In that block, we use a modification of the technique from Section 5.5 for finding a maximum.

12.5 Exercises

You are given a file called class_scores.txt, where each line of the file contains a oneword username and a test score separated by spaces, like below:.

```
GWashington 83
JAdams 86
```

Write code that scans through the file, adds 5 points to each test score, and outputs the usernames and new test scores to a new file, scores2.txt. You are given a file called grades.txt, where each line of the file contains a oneword student username and three test scores separated by spaces, like below:.

```
GWashington 83 77 54
JAdams 86 69 90
```

Write code that scans through the file and determines how many students passed all three tests.

You are given a file called logfile.txt that lists log-on and log-off times for users of a system. A typical line of the file looks like this:

```
Van Rossum, 14:22, 14:37
```

Each line has three entries separated by commas: a username, a log-on time,

and a log-off time. Times are given in 24-hour format. You may assume that all log-ons and log-offs occur within a single workday.

Write a program that scans through the file and prints out all users who were online for at least an hour.

You are given a file called `students.txt`. A typical line in the file looks like:
walter melon melon@email.msmary.edu555-3141

There is a name, an email address, and a phone number, each separated by tabs. Write a program that reads through the file line-by-line, and for each line, capitalizes the first letter of the first and last name and adds the area code 301 to the phone number. Your program should write this to a new file called `students2.txt`. Here is what the first line of the new file should look like:

Walter Melon melon@email.msmary.edu301-555-3141

You are given a file `namelist.txt` that contains a bunch of names. Some of the names are a first name and a last name separated by spaces, like *George Washington*, while others have a middle name, like *John Quincy Adams*. There are no names consisting of just one word or more than three words. Write a program that asks the user to enter initials, like *GW* or *JQA*, and prints all the names that match those initials. Note that initials like *JA* should match both *John Adams* and *John Quincy Adams*.

You are given a file `namelist.txt` that contains a bunch of names. Print out all the names in the list in which the vowels *a*, *e*, *i*, *o*, and *u* appear in order (with repeats possible). The first vowel in the name must be *a* and after the first *u*, it is okay for there to be other vowels. An example is *Ace Elvin Coulson*.

You are given a file called `baseball.txt`. A typical line of the file starts like below. Ichiro Suzuki SEA 162 680 74 ...[more stats]

Each entry is separated by a tab, `\t`. The first entry is the player's name and the second is their team. Following that are 16 statistics. Home runs are the seventh stat and stolen bases are the eleventh. Print out all the players who have at least 20 home runs and at least 20 stolen bases.

For this problem, use the file of NCAA basketball scores as described in Section 12.3.

Find the average of the points scored over all the games in the file. Pick your favorite team and scan through the file to determine how many games they won and how many games they lost.

Find the team(s) that lost by 30 or more points the most times (d) Find all the teams that averaged at least 70 points a game.

(e) Find all the teams that had winning records but were collectively outscored by their opponents. A team is collectively outscored by their opponents if the total number of points the team scored over all their games is less than the total number of points their opponents scored in their games against the team.

Benford's law states that in real data where the values are spread across several orders of magnitude, about 30% of the values will start with the number 1, whereas only about 4.6% of the values will start with the number 9. This is contrary to what we might expect, namely that values starting with 1 and 9 would be equally likely. Using the file `expenses.txt` which consists of a number of costs from an expense account, determine what percentage start with each of the digits 1 through 9. This technique is used by accountants to detect fraud.

Wordplay – Use the file `wordlist.txt` for this problem. Find the following: All words ending in *ime*

All words whose second, third, and fourth letters are *ave*

How many words contain at least one of the letters *r, s, t, l, n, e*

The percentage of words that contain at least one of the letters *r, s, t, l, n, e*

All words with no vowels

All words that contain every vowel

Whether there are more ten-letter words or seven-letter words

The longest word in the list

All palindromes

All words that are words in reverse, like *rat* and *tar*.

Same as above, but only print one word out of each pair.

All words that contain double letters next each other like *aardvark* or *book*, excluding words that end in *lly*

All words that contain a *q* that isn't followed by a *u*
All words that contain *zu* anywhere in the word
All words that contain *ab* in multiple places, like *habitable*
All words with four or more vowels in a row
All words that contain both a *z* and a *w*
All words whose first letter is *a*, third letter is *e* and fifth letter is *i*
All two-letter words

All four-letter words that start and end with the same letter (*u*) All words that contain at least nine vowels.

All words that contain each of the letters *a*, *b*, *c*, *d*, *e*, and *f* in any order. There may be other letters in the word. Two examples are *backfield* and *feedback*.

All words whose first four and last four letters are the same
All words of the form *abcd*dcba*, where *** is arbitrarily long sequence of letters.

All groups of 5 words, like *pat pet pit pot put*, where each word is 3 letters, all words share the same first and last letters, and the middle letter runs through all 5 vowels.

The word that has the most *i*'s.

Write a program to help with word games. The user enters a word and the program uses the wordlist to determine if the user's word is a real word or not.

Suppose we write all the words in the wordlist backwards and then arrange these backwards words alphabetically. Write a program that prints the last word in this modified wordlist.

Print out all combinations of the string *'Python'* plus a three letter English word. Capitalize the first letter of the three letter word. Example combinations are *'PythonCat'*, *'PythonDog'*, and *'PythonTag'*. These are valid combinations because *cat*, *dog*, and *tag* are English words. On the other hand, *'PythonQqz'* would not be a valid combination because *qqz* is not an English word. Use a wordlist to determine which three letter combinations are words.

Write a simple spell-checking program. The user should enter a string and the program should print out a list of all the words it thinks are misspelled. These would be all the words it cannot find in a wordlist.

Crossword cheater: When working on a crossword puzzle, often you will have a word where you know several of the letters, but not all of them. You can write a computer program to help you. For the program, the user should be able to input a word with the letters they know filled in and asterisks for those they don't know. The program should print out a list of all words that fit that description. For example, the input `th***ly` should return all the words that could work, namely *thickly* and *thirdly*.

Ask the user to enter several letters. Then find all the words that can be made with those letters, repeats allowed.

Using the wordlist, produce a dictionary whose keys are the letters *a* through *z* and whose values are the percentage of words that use that letter.

Using the wordlist, produce a dictionary whose keys are the letters *a* through *z* and whose values are the percentage of total letters in the wordlist that are that letter.

Write a program that asks the user for a word and finds all the smaller words that can be made from the letters of that word. The number of occurrences of a letter in a smaller word can't exceed the number of occurrences of the letter in the user's word.

(a) Write a program that reads a file consisting of email addresses, each on its own line. Your program should print out a string consisting of those email addresses separated by semicolons.

(b) Write the same program as above, but the new string should contain only those email addresses that do not end in `@prof.college.edu`.

The file `high_temperatures.txt` contains the average high temperatures for each day of the year in a certain city. Each line of the file consists of the date, written in the month/day format, followed by a space and the average high temperature for that date. Find the 30-day period over which there is the biggest increase in the average high temperature.

In Chapter 6 there was an exercise about the game *Mad Libs*. It asked you to make up a story and leave out some words of the story. Your program should ask the user to enter some words and tell them what types of words to enter. Then print the full story along with the inserted words. Rewrite your program from that exercise to read the story from a file. Reading the story from a file allows people who do not know how to program to use their own stories with the program without having to change the code.

An acronym is an abbreviation that uses the first letter of each word in a phrase. We see them everywhere. For instance, NCAA for National Collegiate Athletic Association or NBC for National Broadcasting Company. Write a program where the user enters an acronym and the program randomly selects words from a wordlist such that the words would fit the acronym. Below is some typical output generated when I ran the program:

```
Enter acronym: ABC
['addressed', 'better', 'common']
Enter acronym: BRIAN
['bank', 'regarding', 'intending', 'army', 'naive']
```

This problem is about a version of the game *Jotto*. The computer chooses a random five-letter word with no repeat letters. The player gets several turns to try to guess the computer's word. On each turn, the player guesses a five-letter word and is told the number of letters that their guess has in common with the computer's word.

The word *part* has the interesting property that if you remove its letters one by one, each resulting step is a real word. For instance, *part* → *pat* → *pa* → *a*. You may remove the letters in any order, and the last (single-letter) word needs to be a real word as well. Find all eight-letter words with this property.

Write a program to cheat at the game *Scrabble*. The user enters a string. Your program should return a list of all the words that can be created from those seven letters.

Functions

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if you find yourself writing the same code

at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

13.1 Basics

Functions are defined with the **def** statement. The statement ends with a colon, and the code that is part of the function is indented below the **def** statement. Here we create a simple function that just prints something.

```
def print_hello():  
    print('Hello!')  
print_hello() print('1234567') print_hello()  
Hello!  
1234567 Hello!
```

The first two lines define the function. In the last three lines we call the function twice.

One use for functions is if you are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a function. For instance, suppose for some reason you need to print a box of stars like the one below at several points in your program.

```
119  
*****  
*  
*  
*****
```

Put the code into a function, and then whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def draw_square():  
    print('*' * 15) print('*', ' '*11, '*') print('*', ' '*11, '*') print('*' * 15)
```

One benefit of this is that if you decide to change the size of the box, you just

have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

13.2 Arguments

We can pass values to functions. Here is an example:

```
def print_hello(n):  
    print('Hello ' * n) print()  
print_hello(3) print_hello(5) times = 2  
print_hello(times)
```

```
Hello Hello Hello  
Hello Hello Hello Hello Hello  
Hello Hello
```

When we call the `print_hello` function with the value 3, that value gets stored in the variable `n`. We can then refer to that variable `n` in our function's code. You can pass more than one value to a function:

```
def multiple_print(string, n):  
    print(string * n) print()  
multiple_print('Hello', 5) multiple_print('A', 10)
```

13.3. RETURNING VALUES

```
HelloHelloHelloHelloHello  
AAAAAAAAAAAA
```

13.3 Returning values

We can write functions that perform calculations and return a result.

Example 1 Here is a simple function that converts temperatures from Celsius to Fahrenheit.

```
def convert(t): return t*9/5+32 print(convert(20))  
68
```

The `return` statement is used to send the result of a function's calculations back to the caller.

Notice that the function itself does not do any printing. The printing is done outside of the function. That way, we can do math with the result, like below.

```
print(convert(20)+5)
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would never be able to do anything with it.

Example 2 As another example, the Python math module contains trig functions, but they only work in radians. Let us write our own sine function that works in degrees.

```
from math import pi, sin
def deg_sin(x):
    return sin(pi*x/180)
```

Example 3 A function can return multiple values as a list.

Say we want to write a function that solves the system of equations $ax + by = e$ and $cx + dy = f$.

It turns out that if there is a unique solution, then it is given by $x = (de - bf)/(ad - bc)$ and $y = (af - ce)/(ad - bc)$. We need our function to return both the x and y solutions.

```
def solve(a,b,c,d,e,f): x = (d*e-b*f)/(a*d-b*c) y = (a*f-c*e)/(a*d-b*c) return [x,y]
```

```
xsol, ysol = solve(2,3,4,1,2,5)
```

```
print('The solution is x = ', xsol, 'and y = ', ysol)
```

The solution is $x = 1.3$ and $y = -0.2$

This method uses the shortcut for assigning to lists that was mentioned in Section 10.3.

Example 4 A `return` statement by itself can be used to end a function early.

```
def multiple_print(string, n, bad_words):
    if string in bad_words:
        return
    print(string * n) print()
```

The same effect can be achieved with an if/else statement, but in some cases, using **return** can make your code simpler and more readable.

13.4 Default arguments and keyword arguments

You can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value. Here is an example:

```
def multiple_print(string, n=1) print(string * n) print()  
multiple_print('Hello', 5) multiple_print('Hello')  
HelloHelloHelloHelloHello  
Hello
```

Default arguments need to come at the end of the function definition, after all of the non-default arguments.

Keyword arguments A related concept to default arguments is *keyword arguments*. Say we have the following function definition:

```
def fancy_print(text, color, background, style, justify):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, Python allows you to name the arguments when calling the function, as shown below:

```
fancy_print(text='Hi', color='yellow', background='black',  
  
style= 'bold', justify='left')  
13.5. LOCAL VARIABLES  
fancy_print(text='Hi', style='bold', justify='left', background='black',  
color='yellow')
```

As we can see, the order of the arguments does not matter when you use keyword arguments.

When defining the function, it would be a good idea to give defaults. For instance, most of the time, the caller would want left justification, a white background, etc. Using these values as defaults means the caller does not have to specify every single argument every time they call the function. Here is an example:

```
def fancy_print(text, color='black', background='white', style='normal',
justify='left'):
```

```
# function code goes here
```

```
fancy_print('Hi', style='bold') fancy_print('Hi', color='yellow',
background='black') fancy_print('Hi')
```

Note We have actually seen default and keyword arguments before—the sep, end and file arguments of the print function.

13.5 Local variables

Let's say we have two functions like the ones below that each use a variable

```
i: def func1():
```

```
for i in range(10):
```

```
print(i)
```

```
def func2(): i=100 func1() print(i)
```

A problem that could arise here is that when we call func1, we might mess up the value of i in func2. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions, and, fortunately, we don't have to worry about this. When a variable is defined inside a function, it is *local* to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

Global variables On the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a *global* variable. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller programs. Here is a short example:

```
def reset():
```

```
global time_left time_left = 0
```

```
def print_time():
```

```
print(time_left) time_left=30
```

In this program we have a variable time_left that we would like multiple functions to have access to. If a function wants to change the value of that variable, we need to tell the function that time_left is a global variable. We use a **global** statement in the function to do this. On the other hand, if we just

want to use the value of the global variable, we do not need a **global** statement.

Arguments We finish the chapter with a bit of a technical detail. You can skip this section for the time being if you don't want to worry about details right now. Here are two simple functions:

```
def func1(x): x = x + 1
def func2(L):
    L = L + [1]
a=3 M=[1,2,3] func1(a) func2(M)
```

When we call `func1` with `a` and `func2` with `L`, a question arises: do the functions change the values of `a` and `L`? The answer may surprise you. The value of `a` is unchanged, but the value of `L` is changed. The reason has to do with a difference in the way that Python handles numbers and lists. Lists are said to be *mutable* objects, meaning they can be changed, whereas numbers and strings are *immutable*, meaning they cannot be changed. There is more on this in Section 19.1.

If we want to reverse the behavior of the above example so that `a` is modified and

`L` is not, do the following:

```
def func1(x): x = x + 1 return x
def func2(L):
    copy = L[:] copy = copy + [1]
a=3 M=[1,2,3] a=func1(a) # note change on this line
```

13.6. EXERCISES `func2(M)`

13.6 Exercises

Write a function called `rectangle` that takes two integers `m` and `n` as arguments and prints out an $m \times n$ box consisting of asterisks. Shown below is the output of `rectangle(2,4)`

```
****
****
```

(a) Write a function called `add_excitement` that takes a list of strings and adds an exclamation point (!) to the end of each string in the list. The program should modify the original list and not return anything.

(b) Write the same function except that it should not modify the original list and should instead return a new list.

Write a function called `sum_digits` that is given an integer `num` and returns the sum of the digits of `num`.

The *digital root* of a number n is obtained as follows: Add up the digits n to get a new number. Add up the digits of that to get another new number. Keep doing this until you get a number that has only one digit. That number is the digital root.

For example, if $n = 45893$, we add up the digits to get $4+5+8+9+3 = 29$. We then add up the digits of 29 to get $2+9 = 11$. We then add up the digits of 11 to get $1+1 = 2$. Since 2 has only one digit, 2 is our digital root.

Write a function that returns the digital root of an integer n . [Note: there is a shortcut, where the digital root is equal to $n \bmod 9$, but do not use that here.]

Write a function called `first_diff` that is given two strings and returns the first location in which the strings differ. If the strings are identical, it should return -1.

Write a function called `binom` that takes two integers n and k and returns the binomial coefficient $\binom{n}{k}$. The definition is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Write a function that takes an integer n and returns a random integer with exactly n digits. For instance, if n is 3, then 125 and 593 would be valid return values, but 093 would not because that is really 93, which is a two-digit number.

Write a function called `number_of_factors` that takes an integer and returns how many factors the number has.

Write a function called `factors` that takes an integer and returns a list of its factors.

Write a function called `closest` that takes a list of numbers `L` and a number `n` and returns the largest element in `L` that is not larger than `n`. For instance, if `L=[1,6,3,9,11]` and `n=8`, then the function should return 6, because 6 is the closest thing in `L` to 8 that is not larger than 8. Don't worry about if all of the things in `L` are smaller than `n`.

Write a function called `matches` that takes two strings as arguments and returns how many matches there are between the strings. A match is where the two strings have the same character at the same index. For instance, `'python'` and `'path'` match in the first, third, and fourth characters, so the function should return 3.

Recall that if `s` is a string, then `s.find('a')` will find the location of the *first* `a` in `s`. The problem is that it does not find the location of every `a`. Write a function called `findall` that given a string and a single character, returns a list containing all of the locations of that character in the string. It should return an empty list if there are no occurrences of the character in the string.

Write a function called `change_case` that given a string, returns a string with each upper case letter replaced by a lower case letter and vice-versa.

Write a function called `is_sorted` that is given a list and returns **True** if the list is sorted and **False** otherwise.

Write a function called `root` that is given a number `x` and an integer `n` and returns $x^{1/n}$. In the function definition, set the default value of `n` to 2.

Write a function called `one_away` that takes two strings and returns **True** if the strings are of the same length and differ in exactly one letter, like `bike/hike` or `water/wafer`.

(a) Write a function called `primes` that is given a number `n` and returns a list of the first `n` primes. Let the default value of `n` be 100.

(b) Modify the function above so that there is an optional argument called `start` that allows the list to start at a value other than 2. The function should return the first `n` primes that are greater than or equal to `start`. The default value of `start` should be 2.

Our number system is called *base 10* because we have ten digits: 0, 1, ..., 9.

Some cultures, including the Mayans and Celts, used a base 20 system. In one version of this system, the 20 digits are represented by the letters A through T. Here is a table showing a few conversions:

10	20	10	20	10	20	10	20
0	A	8	I	16	Q	39	BT
1	B	9	J	17	R	40	CA
2	C	10	K	18	S	41	CB
3	D	11	L	19	T	60	DA
4	E	12	M	20	BA	399	TT
5	F	13	N	21	BB	400	BAA
6	G	14	O	22	BC	401	BAB
7	H	15	P	23	BD	402	BAC

Write a function called `base20` that converts a base 10 number to base 20. It should return the result as a string of base 20 digits. One way to convert is to find the remainder when the number is divided by 20, then divide the number by 20, and repeat the process until the number is 0. The remainders are the base 20 digits in reverse order, though you have to convert them into their letter equivalents.

13.6. EXERCISES

Write a function called `verbose` that, given an integer less than 10^{15} , returns the name of the integer in English. As an example, `verbose(123456)` should return one hundred twenty-threethousand,fourhundredfifty-six.

Write a function called `merge` that takes two already sorted lists of possibly different lengths, and merges them into a single sorted list.

Do this using the `sort` method.

Do this without using the `sort` method.

In Chapter 12, the way we checked to see if a word `w` was a real word was: `if w in words:`

where `words` was the list of words generated from a wordlist. This is unfortunately slow, but there is a faster way, called a *binary search*. To implement a binary search in a function, start by comparing `w` with the

middle entry in words. If they are equal, then you are done and the function should return **True**. On the other hand, if *w* comes before the middle entry, then search the first half of the list. If it comes after the middle entry, then search the second half of the list. Then repeat the process on the appropriate half of the list and continue until the word is found or there is nothing left to search, in which case the function should return **False**. The *<* and *>* operators can be used to alphabetically compare two strings.

A Tic-tac-toe board can be represented by a 3×3 two-dimensional list, where zeroes stand for empty cells, ones stand for *X*'s and twos stand for *O*'s. Write a function that is given such a list and randomly chooses a spot in which to place a 2. The spot chosen must currently be a 0 and a spot must be chosen. Write a function that is given such a list and checks to see if someone has won. Return **True** if there is a winner and **False** otherwise.

Write a function that is given a 9 × 9 potentially solved Sudoku and returns **True** if it is solved correctly and **False** if there is a mistake. The Sudoku is correctly solved if there are

no repeated numbers in any row or any column or in any of the nine “blocks.”

Object-Oriented Programming

About a year or so after I started programming, I decided to make a game to play *Wheel of Fortune*. I wrote the program in the BASIC programming language and it got to be pretty large, a couple thousand lines. It mostly worked, but whenever I tried to fix something, my fix would break something in a completely different part of the program. I would then fix that and break something else. Eventually I got the program working, but after a while I was afraid to even touch it.

The problem with the program was that each part of the program had access to the variables from the other parts. A change of a variable in one part would mess up things in the others. One solution to this type of problem is *object-oriented programming*. One of its chief benefits is *encapsulation*, where you divide your program into pieces and each piece internally operates independently of the others. The pieces interact with each other, but they don't need to know exactly how each one accomplishes its tasks. This requires some planning and set-up time before you start your program, and so

it is not always appropriate for short programs, like many of the ones that we have written so far.

We will just cover the basics of object-oriented programming here. Object-oriented programming is used extensively in software design and I would recommend picking up another book on programming or software design to learn more about designing programs in an object-oriented way.

14.1 Python is object-oriented

Python is an object-oriented programming language, and we have in fact been using many object-oriented concepts already. The key notion is that of an *object*. An object consists of two things: data and functions (called *methods*) that work with that data. As an example, strings in Python are objects. The data of the string object is the actual characters that make up that string. The methods are things like `lower`, `replace`, and `split`. In Python, everything is an object. That includes not only strings and lists, but also integers, floats, and even functions themselves.

129

14.2 Creating your own classes

A *class* is a template for objects. It contains the code for all the object's methods.

A simple example Here is a simple example to demonstrate what a class looks like. It does not do anything interesting.

```
class Example:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def add(self):
        return self.a + self.b
e = Example(8, 6)
print(e.add())
```

To create a class, we use the `class` statement. Class names usually start with a capital.

Most classes will have a method called `__init__`. The underscores indicate that it is a special kind of method. It is called a *constructor*, and it is

automatically called when someone creates a new object from your class. The constructor is usually used to set up the class's variables. In the above program, the constructor takes two values, a and b, and assigns the class variables a and b to those values.

The first argument to every method in your class is a special variable called `self`. Every time your class refers to one of its variables or methods, it must precede them by `self`. The purpose of `self` is to distinguish your class's variables and methods from other variables and functions in the program.

To create a new object from the class, you call the class name along with any values that you want to send to the constructor. You will usually want to assign it to a variable name. This is what the line `e=Example(8,6)` does.

To use the object's methods, use the dot operator, as in `e.addmod()`.

A more practical example Here is a class called `Analyzer` that performs some simple analysis on a string. There are methods to return how many words are in the string, how many are of a given length, and how many start with a given string.

```
from string import punctuation
class Analyzer:
    def __init__(self, s):
        for c in punctuation: s = s.replace(c, "")
```

```
14.2. CREATING YOUR OWN CLASSES
s = s.lower()
self.words = s.split()
def number_of_words(self): return len(self.words)
def starts_with(self, s):
    return len([w for w in self.words if w[:len(s)]==s])
```

```
def number_with_length(self, n): return len([w for w in self.words if
len(w)==n])
```

```
s = "This is a test of the class."
analyzer = Analyzer(s)
print(analyzer.words)
print('Number of words:', analyzer.number_of_words())
print('Number of words starting with "t":', analyzer.starts_with('t'))
print('Number of 2-letter words:', analyzer.number_with_length(2))
```

```
['this', 'is', 'a', 'test', 'of', 'the', 'class']
```

```
Number of words: 7
```

```
Number of words starting with "t": 3
```

```
Number of 2-letter words: 2
```

```
A few notes about this program:
```

One reason why we would wrap this code up in a class is we can then use it a variety of different programs. It is also good just for organizing things. If all our program is doing is just analyzing some strings, then there's not too much of a point of writing a class, but if this were to be a part of a larger program, then using a class provides a nice way to separate the Analyzer code from the rest of the code. It also means that if we were to change the internals of the Analyzer class, the rest of the program would not be affected as long as the interface, the way the rest of the program interacts with the class, does not change. Also, the Analyzer class can be imported as-is in other programs.

The following line accesses a class variable:

```
print(analyzer.words)
```

You can also change class variables. This is not always a good thing. In some cases this is convenient, but you have to be careful with it. Indiscriminate use of class variables goes against the idea of encapsulation and can lead to programming errors that are hard to fix. Some other object-oriented programming languages have a notion of public and private variables, public variables being those that anyone can access and change, and private variables being only accessible to methods within the class. In Python all variables are public, and it is up to the programmer to be responsible with them. There is a convention where you name those variables that you want to be private with a starting underscore, like `_var1`. This serves to let others know that this variable is internal to the class and shouldn't be touched.

14.3 Inheritance

In object-oriented programming there is a concept called *inheritance* where you can create a class that builds off of another class. When you do this, the new class gets all of the variables and methods of the class it is inheriting from (called the *base class*). It can then define additional variables and methods that are not present in the base class, and it can also *override* some

of the methods of the base class. That is, it can rewrite them to suit its own purposes. Here is a simple example:

```
class Parent:
def __init__(self, a): self.a = a
def method1(self): return self.a*2
def method2(self): return self.a+'!!!'
class Child(Parent):
def __init__(self, a, b):
self.a = a self.b = b
def method1(self): return self.a*7
def method3(self): return self.a + self.b
p = Parent('hi') c = Child('hi', 'bye')

print ('Parent method 1: ', p.method1()) print('Parent method 2: ',
p.method2()) print() print('Child method 1: ', c.method1()) print('Child
method 2: ', c.method2()) print('Child method 3: ', c.method3())
```

```
Parent method 1: hihi Parent method 2: hi!!!
Child method 1: hihihihihihih Child method 2: hi!!!
Child method 3: hibye
```

We see in the example above that the child has overridden the parent's method1, causing it to now repeat the string seven times. The child has inherited the parent's method2, so it can use it without having to define it. The child also adds some features to the parent class, namely a new variable b and a new method, method3.

A note about syntax: when inheriting from a class, you indicate the parent class in parentheses in the **class** statement.

14.4. A PLAYING-CARD EXAMPLE

If the child class adds some new variables, it can call the parent class's constructor as demonstrated below. Another use is if the child class just wants to add on to one of the parent's methods. In the example below, the child's print_var method calls the parent's print_var method and adds an additional line.

```

class Parent:
def __init__(self, a): self.a = a

def print_var(self): print("The value of this class's variables are:")
print(self.a)

class Child(Parent):
def __init__(self, a, b): Parent.__init__(self, a) self.b = b def print_var(self):
Parent.print_var(self) print(self.b)

```

Note You can also inherit from Python built-in types, like strings (**str**) and lists (**list**), as well as any classes defined in the various modules that come with Python. **Note** Your code can inherit from more than one class at a time, though this can be a little tricky.

14.4 A playing-card example

In this section we will show how to design a program with classes. We will create a simple hi-lo card game where the user is given a card and they have to say if the next card will be higher or lower than it. This game could easily be done without classes, but we will create classes to represent a card and a deck of cards, and these classes can be reused in other card games.

We start with a class for a playing card. The data associated with a card consists of its value (2 through 14) and its suit. The Card class below has only one method, `__str__`. This is a special method that, among other things, tells the **print** function how to print a Card object.

```

class Card:
def __init__(self, value, suit):
self.value = value self.suit = suit
def __str__(self):
names = ['Jack', 'Queen', 'King', 'Ace'] if self.value <= 10:
return '{} of {}'.format(self.value, self.suit)
else:
return '{} of {}'.format(names[self.value-11], self.suit)

```

Next we have a class to represent a group of cards. Its data consists of a list of Card objects. It has a number of methods: `nextCard` which removes the

first card from the list and returns it; hasCard which returns **True** or **False** depending on if there are any cards left in the list; size, which returns how many cards are in the list; and shuffle, which shuffles the list.

```
import random
class Card_group:
    def __init__(self, cards=[]):
        self.cards = cards
    def nextCard(self): return self.cards.pop(0)
    def hasCard(self): return len(self.cards)>0
    def size(self): return len(self.cards)
    def shuffle(self):
        random.shuffle(self.cards)
```

We have one more class Standard_deck, which inherits from Card_group. The idea here is that Card_group represents an arbitrary group of cards, and Standard_deck represents a specific group of cards, namely the standard deck of 52 cards used in most card games.

```
class Standard_deck(Card_group):
    def __init__(self):
        self.cards = []
        for s in ['Hearts', 'Diamonds', 'Clubs', 'Spades']:
            for v in range(2,15):
                self.cards.append(Card(v, s))
```

Suppose we had just created a single class that represented a standard deck along with all the common operations like shuffling. If we wanted to create a new class for a Pinochle game or some other game that doesn't use the standard deck, then we would have to copy and paste the standard deck code and modify lots of things. By doing things more generally, like we've done here, each time we want a new type of deck, we can build off of (inherit from) what is in Card_group. For instance, a Pinochle deck class would look like this:

```
class Pinochle_deck(Card_group):
    def __init__(self):
        self.cards = []
        for s in ['Hearts', 'Diamonds', 'Clubs', 'Spades']*2:
```

14.4. A PLAYING-CARD EXAMPLE

```
for v in range(9,15):
    self.cards.append(Card(v, s))
```

A Pinochle deck has only nines, tens, jacks, queens, kings, and aces. There are two copies of each card in each suit.

Here is the hi-low program that uses the classes we have developed here. One way to think of what we have done with the classes is that we have built up a miniature card programming language, where we can think about how a card game works and not have to worry about exactly how cards are shuffled or dealt or whatever, since that is wrapped up into the classes. For the hi-low game, we get a new deck of cards, shuffle it, and then deal out the cards one at a time. When we run out of cards, we get a new deck and shuffle it. A nice feature of this game is that it deals out all 52 cards of a deck, so a player can use their memory to help them play the game.

```
deck = Standard_deck() deck.shuffle()
```

```
new_card = deck.nextCard() print('\n', new_card) choice = input("Higher (h) or lower (l): ") streak = 0
```

```
while (choice=='h' or choice=='l):
```

```
if not deck.hasCard(): deck = Standard_deck() deck.shuffle()
```

```
old_card = new_card new_card = deck.nextCard()
```

```
if (choice.lower()=='h' and new_card.value>old_card.value or\
choice.lower()=='l' and new_card.value<old_card.value):
```

```
    streak = streak + 1 print("Right! That's", streak, "in a row!")
```

```
elif (choice.lower()=='h' and new_card.value<old_card.value or\
choice.lower()=='l' and new_card.value>old_card.value):
```

```
    streak = 0
```

```
    print('Wrong.')
```

```
else:
```

```
    print('Push.')
```

```
    print('\n', new_card) choice = input("Higher (h) or lower (l): ")
```

King of Clubs

Higher (h) or lower (l): l Right! That's 1 in a row!

2 of Spades

Higher (h) or lower (l): h Right! That's 2 in a row!

14.5 A Tic-tac-toe example

In this section we create an object-oriented Tic-tac-toe game. We use a class to wrap up the logic of the game. The class contains two variables, an integer representing the current player, and a 3×3 list representing the board. The board variable consists of zeros, ones, and twos. Zeros represent an open spot, while ones and twos represent spots marked by players 1 and 2, respectively. There are four methods:

`get_open_spots` — returns a list of the places on the board that have not yet been marked by players

`is_valid_move`— takes a row and a column representing a potential move, and returns

True if move is allowed and **False** otherwise

`make_move`— takes a row and a column representing a potential move, calls `is_valid_move` to see if the move is okay, and if it is, sets the board array accordingly and changes the player

`check_for_winner` — scans through the board list and returns 1 if player 1 has won, 2 if player 2 has won, 0 if there are no moves remaining and no winner, and -1 if the game should continue

Here is the code for the class:

```
class tic_tac_toe:
    def __init__(self):
        self.B = [[0,0,0], [0,0,0],
                  [0,0,0]]
        self.player = 1
    def get_open_spots(self):
        return [[r,c] for r in range(3) for c in range(3) if self.B[r][c]==0]
    def is_valid_move(self,r,c):
        if 0<=r<=2 and 0<=c<=2 and self.B[r][c]==0:
            return True
```

```

return False
def make_move(self,r,c):
if self.is_valid_move(r,c):
self.B[r][c] = self.player self.player = (self.player+2)%2 + 1 def
check_for_winner(self):

```

14.5. A TIC-TAC-TOE EXAMPLE

```

for c in range(3):
if self.B[0][c]==self.B[1][c]==self.B[2][c]!=0:
return self.B[0][c] for r in range(3):
if self.B[r][0]==self.B[r][1]==self.B[r][2]!=0: return self.B[r][0] if self.B[0]
[0]==self.B[1][1]==self.B[2][2]!=0: return self.B[0][0] if self.B[2]
[0]==self.B[1][1]==self.B[0][2]!=0:
return self.B[2][0]
if self.get_open_spots()==[]: return 0
return -1

```

This class consists of the logic of the game. There is nothing in the class that is specific to the user interface. Below we have a text-based interface using print and input statements. If we decide to use a graphical interface, we can use the Tic_tac_toe class without having to change anything about it. Note that the get_open_spots method is not used by this program. It is useful, however, if you want to implement a computer player. A simple computer player would call that method and use random.choice method to choose a random element from the returned list of spots.

```

def print_board():
chars = ['-','X','O'] for r in range(3):
for c in range(3):
print(chars[game.B[r][c]], end=' ')
print()
game = tic_tac_toe() while game.check_for_winner()!=-1:

print_board() r,c = eval(input('Enter spot, player ' + str(game.player) + ': '))
game.make_move(r,c)

print_board() x = game.check_for_winner() if x==0: print("It's a draw.")
else:

```

```
print('Player', x, 'wins!')
```

Here is what the first couple of turns look like:

```
-  
-  
-
```

Enter spot, player 1: 1,1

```
-
```

X

```
-
```

Enter spot, player 2: 0,2

- O

X

- -Enter spot, player 1:

14.6 Further topics

Special methods — We have seen two special methods already, the constructor `__init__` and the method `__str__` which determines what your objects look like when printed. There are many others. For instance, there is `__add__` that allows your object to use the `+` operator. There are special methods for all the Python operators. There is also a method called `__len__` which allows your object to work with the built in `len` function. There is even a special method, `__getitem__` that lets your program work with list and string brackets `[]`.

Copying objects — If you want to make a copy of an object `x`, it is not enough to do the following:

```
x_copy = x
```

The reason is discussed in Section [19.1](#). Instead, do the following:

```
from copy import copy x_copy = copy(x)
```

Keeping your code in multiple files — If you want to reuse a class in several programs, you do not have to copy and paste the code into each. You can save it in a file and use an import statement to import it into your programs. The file will need to be somewhere your program can find it, like in the same directory.

from analyzer **import** Analyzer

14.7 Exercises

Write a class called Investment with fields called principal and interest. The constructor should set the values of those fields. There should be a method called value_after that returns the value of the investment after n years. The formula for this is $p(1 + i)^n$, where p is the principal, and i is the interest rate. It should also use the special method `__str__` so that printing the object will result in something like below:

Principal - \$1000.00, Interest rate - 5.12%

Write a class called Product. The class should have fields called name, amount, and price, holding the product's name, the number of items of that product in stock, and the regular price of the product. There should be a method get_price that receives the number of items to be bought and returns a the cost of buying that many items, where the regular price

14.7. EXERCISES

is charged for orders of less than 10 items, a 10% discount is applied for orders of between 10 and 99 items, and a 20% discount is applied for orders of 100 or more items. There should also be a method called make_purchase that receives the number of items to be bought and decreases amount by that much.

Write a class called Password_manager. The class should have a list called old_passwords that holds all of the user's past passwords. The last item of the list is the user's current password. There should be a method called get_password that returns the current password and a method called set_password that sets the user's password. The set_password method should only change the password if the attempted password is different from all the user's past passwords. Finally, create a method called is_correct that receives a string and returns a boolean **True** or **False** depending on whether the string is equal to the current password or not.

Write a class called Time whose only field is a time in seconds. It should have a method called convert_to_minutes that returns a string of minutes and seconds formatted as in the following example: if seconds is 230, the method

should return `'5:50'`. It should also have a method called `convert_to_hours` that returns a string of hours, minutes, and seconds formatted analogously to the previous method.

Write a class called `Wordplay`. It should have a field that holds a list of words. The user of the class should pass the list of words they want to use to the class. There should be the following methods:

`words_with_length(length)` — returns a list of all the words of length `length`
`starts_with(s)`— returns a list of all the words that start with `s`
`ends_with(s)`— returns a list of all the words that end with `s`
`palindromes()`— returns a list of all the palindromes in the list
`only(L)` — returns a list of the words that contain only those letters in `L`
`avoids(L)` — returns a list of the words that contain none of the letters in `L`

Write a class called `Converter`. The user will pass a length and a unit when declaring an object from the class—for example, `c=Converter(9,'inches')`. The possible units are inches, feet, yards, miles, kilometers, meters, centimeters, and millimeters. For each of these units there should be a method that returns the length converted into those units. For example, using the `Converter` object created above, the user could call `c.feet()` and should get 0.75 as the result.

Use the `Standard_deck` class of this section to create a simplified version of the game *War*. In this game, there are two players. Each starts with half of a deck. The players each deal the top card from their decks and whoever has the higher card wins the other player's cards and adds them to the bottom of his deck. If there is a tie, the two cards are eliminated from play (this differs from the actual game, but is simpler to program). The game ends when one player runs out of cards.

Write a class that inherits from the `Card_group` class of this chapter. The class should represent a deck of cards that contains only hearts and spaces, with only the cards 2 through 10 in each suit. Add a method to the class called `next2` that returns the top two cards from the deck.

Write a class called `Rock_paper_scissors` that implements the logic of the game Rockpaper-scissors. For this game the user plays against the computer for a certain number of rounds. Your class should have fields for the how

many rounds there will be, the current round number, and the number of wins each player has. There should be methods for getting the computer's choice, finding the winner of a round, and checking to see if someone has one the (entire) game. You may want more methods.

(a) Write a class called `Connect4` that implements the logic of a Connect4 game. Use the `Tic_tac_toe` class from this chapter as a starting point.

(b) Use the `Connect4` class to create a simple text-based version of the game. Write a class called `Poker_hand` that has a field that is a list of `Card` objects.

There should be the following self-explanatory methods:

`has_royal_flush`, `has_straight_flush`, `has_four_of_a_kind`, `has_full_house`, `has_flush`, `has_straight`, `has_three_of_a_kind`, `has_two_pair`, `has_pair`

There should also be a method called `best` that returns a string indicating what the best hand is that can be made from those cards.

Chapter 15

GUI Programming with Tkinter

Up until now, the only way our programs have been able to interact with the user is through keyboard input via the **input** statement. But most real programs use windows, buttons, scrollbars, and various other things. These *widgets* are part of what is called a *Graphical User Interface* or GUI. This chapter is about GUI programming in Python with Tkinter.

All of the widgets we will be looking at have far more options than we could possibly cover here. An excellent reference is Fredrik Lundh's *Introduction to Tkinter* [2].

15.1 Basics

Nearly every GUI program we will write will contain the following three lines: **from** tkinter **import** * `root = Tk()` `mainloop()`

The first line imports all of the GUI stuff from the tkinter module. The second line creates a window on the screen, which we call `root`. The third line puts the program into what is essentially a long-running while loop called the *event loop*. This loop runs, waiting for keypresses, button clicks, etc., and it exits when the user closes the window.

Here is a working GUI program that converts temperatures from Fahrenheit to Celsius.

```
from tkinter import *
def calculate():

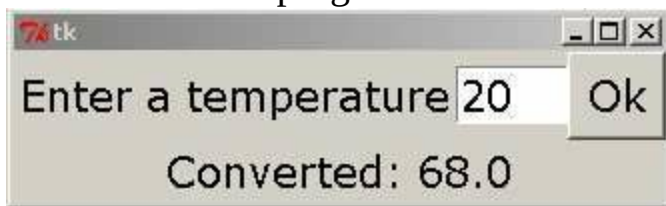
temp = int(entry.get()) temp = 9/5*temp+32 output_label.configure(text =
'Converted: {:.1f}'.format(temp)) entry.delete(0,END)

143 root = Tk() message_label = Label(text='Enter a temperature', font=
('Verdana', 16))

output_label = Label(font=( 'Verdana', 16)) entry = Entry(font=('Verdana',
16), width=4) calc_button = Button(text='Ok', font=('Verdana', 16),
command=calculate)

message_label.grid(row=0, column=0) entry.grid(row=0, column=1)
calc_button.grid(row=0, column=2) output_label.grid(row=1, column=0,
columnspan=3) mainloop()
```

Here is what the program looks like:



We now will examine the components of the program separately.

15.2 Labels

A label is a place for your program to place some text on the screen. The following code creates a label and places it on the screen.

```
hello_label = Label(text='hello') hello_label.grid(row=0, column=0)
```

We call Label to create a new label. The capital L is required. Our label's name is hello_label. Once created, use the grid method to place the label on the screen. We will explain grid in the next section.

Options There are a number of options you can change including font size

and color. Here are some examples:

```
hello_label = Label(text= 'hello', font=('Verdana', 24, 'bold'), bg='blue',  
fg='white')
```

Note the use of keyword arguments. Here are a few common options:

- **font** — The basic structure is `font= (font name, font size, style)`. You can leave out the font size or the style. The choices for style are **'bold'**, **'italic'**, **'underline'**, **'overstrike'**, **'roman'**, and **'normal'** (which is the default). You can combine multiple styles like this:

'bold italic'.

15.3. GRID

fg and bg — These stand for foreground and background. Many common color names can be used, like **'blue'**, **'green'**, etc. Section 16.2 describes how to get essentially any color.

width — This is how many characters long the label should be. If you leave this out, Tkinter will base the width off of the text you put in the label. This can make for unpredictable results, so it is good to decide ahead of time how long you want your label to be and set the width accordingly.

height — This is how many rows high the label should be. You can use this for multiline labels. Use newline characters in the text to get it to span multiple lines. For example, `text='hi\nthere'`.

There are dozens more options. The aforementioned *Introduction to Tkinter* [2] has a nice list of the others and what they do.

Changing label properties Later in your program, after you've created a label, you may want to change something about it. To do that, use its `configure` method. Here are two examples that change the properties of a label called `label`:

```
label.configure(text='Bye') label.configure(bg='white', fg='black')
```

Setting text to something using the `configure` method is kind of like the GUI

equivalent of a **print** statement. However, in calls to configure we cannot use commas to separate multiple things to print. We instead need to use string formatting. Here is a **print** statement and its equivalent using the configure method.

```
print ('a =', a, 'and b =', b)
```

```
label.configure(text='a = {}, and b = {}'.format(a,b))
```

The configure method works with most of the other widgets we will see. 15.3 grid

The grid method is used to place things on the screen. It lays out the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

```
(row=0, column=0) (row=0, column=1) (row=0, column=2)
(row=1, column=0) (row=1, column=1) (row=1, column=2)
(row=2, column=0) (row=2, column=1) (row=2, column=2)
```

Spanning multiple rows or columns There are optional arguments, rowspan and colspan, that allow a widget to take up more than one row or column. Here is an example of several grid statements followed by what the layout will look like:

```
label1.grid(row=0, column=0) label2.grid(row=0, column=1)
label3.grid(row=1, column=0, colspan=2) label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```

```
label1 label2
label 3 label4
label5
```

Spacing To add extra space between widgets, there are optional arguments padx and pady.

Important note Any time you create a widget, to place it on the screen you need to use grid (or one of its cousins, like pack, which we will talk about later). Otherwise it will not be visible.

15.4 Entry boxes

Entry boxes are a way for your GUI to get text input. The following example creates a simple entry box and places it on the screen.

```
entry = Entry() entry.grid(row=0, column=0)
```

Most of the same options that work with labels work with entry boxes (and most of the other widgets we will talk about). The width option is particularly helpful because the entry box will often be wider than you need.

Getting text To get the text from an entry box, use its get method. This will return a string. If you need numerical data, use **eval** (or **int** or **float**) on the string. Here is a simple example that gets text from an entry box named entry.

```
string_value = entry.get() num_value = eval(entry.get())
```

Deleting text To clear an entry box, use the following:

```
entry.delete(0,END)
```

Inserting text To insert text into an entry box, use the following:

```
entry.insert(0, 'hello')
```

15.5 Buttons

The following example creates a simple button:

15.5. BUTTONS

```
ok_button = Button(text='Ok')
```

To get the button to do something when clicked, use the command argument. It is set to the name of a function, called a *callback function*. When the button is clicked, the callback function is called. Here is an example:

```
from tkinter import *
```

```
def callback(): label.configure(text='Button clicked')
```

```
root = Tk() label = Label(text='Not clicked') button = Button(text='Click me',  
command=callback)
```

```
label.grid(row=0, column=0) button.grid(row=1, column=0) mainloop()
```

When the program starts, the label says Click me. When the button is clicked, the callback function callback is called, which changes the label to say Button clicked.



lambda trick Sometimes we will want to pass information to the callback function, like if we have several buttons that use the same callback function and we want to give the function information about which button is being clicked. Here is an example where we create 26 buttons, one for each letter of the alphabet. Rather than use 26 separate Button() statements and 26 different functions, we use a list and one function.

```
from tkinter import * alphabet =
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
def callback(x): label.configure(text='Button {}
clicked'.format(alphabet[x])) root = Tk()
```

```
label = Label() label.grid(row=1, column=0, columnspan=26)
buttons = [0]*26 # create a list to hold 26 buttons for i in range(26):
```

```
buttons[i] = Button(text=alphabet[i], command = lambda x=i:
callback(x))
```

buttons[i].grid(row=0, column=i) mainloop() We note a few things about this program. First, we set buttons=[0]*26. This creates a list with 26 things in it. We don't really care what those things are because they will be replaced with buttons. An alternate way to create the list would be to set buttons=[] and use the append method.



We only use one callback function and it has one argument, which indicates which button was clicked. As far as the **lambda** trick goes, without getting into the details, command=callback(i) does not work, and that is why we resort to the **lambda** trick. You can read more about **lambda** in Section 23.2. An alternate approach is to use classes.

15.6 Global variables

Let's say we want to keep track of how many times a button is clicked. An easy way to do this is to use a global variable as shown below.

```
from tkinter import *
def callback():
    global num_clicks
    num_clicks = num_clicks + 1
    label.configure(text='Clicked {} times.'.format(num_clicks))
num_clicks = 0
root = Tk()
label = Label(text='Not clicked')
button = Button(text='Click me',
    command=callback)
label.grid(row=0, column=0)
button.grid(row=1, column=0)
mainloop()
```



We will be using a few global variables in our GUI programs. Using global variables unnecessarily, especially in long programs, can cause difficult to find errors that make programs hard to maintain, but in the short programs that we will be writing, we should be okay. Object-oriented programming provides an alternative to global variables.

15.7 Tic-tac-toe

Using Tkinter, in only about 20 lines we can make a working tic-tac-toe program:

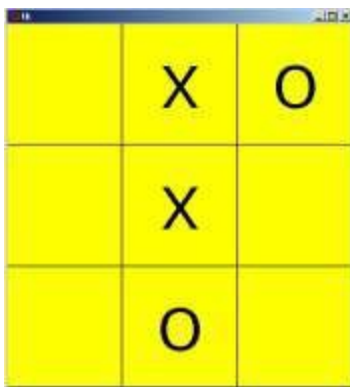
```
from tkinter import *
def callback(r,c):
    global player
    if player == 'X':
        b[r][c].configure(text = 'X')
        player = 'O'
    else :
        b[r][c].configure(text = 'O')
        player = 'X'
root = Tk()
b = [[0,0,0], [0,0,0], [0,0,0]]
for i in range(3):
    for j in range(3):
```

```

b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow', command =
lambda r=i,c=j: callback(r,c))
b[i][j].grid(row = i, column = j)
player = 'X'
mainloop()

```

The program works, though it does have a few problems, like letting you change a cell that already has something in it. We will fix this shortly. First, let's look at how the program does what it does. Starting at the bottom, we have a variable `player` that keeps track of whose turn it is. Above that we create the board, which consists of nine buttons stored in a two-dimensional list. We use the **lambda** trick to pass the row and column of the clicked button to the callback function. In the callback function we write an `X` or an `O` into the button that was clicked and change the value of the global variable `player`.



Correcting the problems To correct the problem about being able to change a cell that already has something in it, we need to have a way of knowing which cells have `X`'s, which have `O`'s, and which are empty. One way is to use a Button method to ask the button what its text is. Another way, which we will do here is to create a new two-dimensional list, which we will call `states`, that will keep track of things. Here is the code.

```

from tkinter import *
def callback(r,c):
    global player
    if player == 'X' and states[r][c] == 0:
        b[r][c].configure(text='X')
        states[r][c] = 'X'
        player = 'O'
    if player == 'O' and states[r][c] == 0:

```

```

b[r][c].configure(text='O') states[r][c] = 'O' player = 'X'
root = Tk()
states = [[0,0,0], [0,0,0],
[0,0,0]]
b = [[0,0,0], [0,0,0],
[0,0,0]]
for i in range(3):
    for j in range(3):

b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow', command =
lambda r=i,c=j: callback(r,c))

b[i][j].grid(row = i, column = j)
player = 'X' mainloop()

```

We have not added much to the program. Most of the new action happens in the callback function. Every time someone clicks on a cell, we first check to see if it is empty (that the corresponding index in states is 0), and if it is, we display an X or O on the screen and record the new value in states. Many games have a variable like states that keeps track of what is on the board.

Checking for a winner We have a winner when there are three X's or three O's in a row, either vertically, horizontally, or diagonally. To check if there are three in a row across the top row, we can use the following if statement:

```

if states[0][0]==states[0][1]==states[0][2]!=0:
    stop_game=True b[0][0].configure(bg='grey') b[0][1].configure(bg='grey')
    b[0][2].configure(bg='grey')

```

This checks to see if each of the cells has the same nonzero entry. We are using the shortcut from Section 10.3 here in the if statement. There are more verbose if statements that would work. If we do find a winner, we highlight the winning cells and then set a global variable stop_game equal to **True**. This variable will be used in the callback function. Whenever the variable is **True** we should not allow any moves to take place.

Next, to check if there are three in a row across the middle row, change the first coordinate from 0 to 1 in all three references, and to check if there are three in a row across the bottom, change the 0's to 2's. Since we will have

three very similar if statements that only differ in one location, a for loop can be used to keep the code short:

```
for i in range(3):
    if states[i][0]==states[i][1]==states[i][2]!=0:

        b[i][0].configure(bg='grey') b[i][1].configure(bg='grey')
        b[i][2].configure(bg='grey') stop_game = True
```

Next, checking for vertical winners is pretty much the same except we vary the second coordinate instead of the first. Finally, we have two further if statements to take care of the diagonals. The full program is at the end of this chapter. We have also added a few color options to the configure statements to make the game look a little nicer.

Further improvements From here it would be easy to add a restart button. The callback function for that variable should set stop_game back to false, it should set states back to all zeroes, and it should configure all the buttons back to text="" and bg='yellow'.

To add a computer player would also not be too difficult, if you don't mind it being a simple computer player that moves randomly. That would take about 10 lines of code. To make an intelligent computer player is not too difficult. Such a computer player should look for two O's or X's in a row in order to try to win or block, as well avoid getting put into a no-win situation.

```
from tkinter import *
def callback(r,c):
    global player
    if player == 'X' and states[x][y] == 0 and stop_game==False:

        b[r][c].configure(text='X', fg='blue', bg='white') states[r][c] = 'X' player = 'O'
        if player == 'O' and states[r][c] == 0 and stop_game==False: b[r]
        [c].configure(text='O', fg='orange', bg='black') states[r][c] = 'O' player = 'X'
        check_for_winner()

def check_for_winner():
    global stop_game for i in range(3):
```

```

if states[i][0]==states[i][1]==states[i][2]!=0:

b[i][0].configure(bg='grey') b[i][1].configure(bg='grey')
b[i][2].configure(bg='grey') stop_game = True

for i in range(3):
if states[0][i]==states[1][i]==states[2][i]!=0:

b[0][i].configure(bg='grey') b[1][i].configure(bg='grey')
b[2][i].configure(bg='grey') stop_game = True
if states[0][0]==states[1][1]==states[2][2]!=0:
b[0][0].configure(bg='grey') b[1][1].configure(bg='grey')
b[2][2].configure(bg='grey') stop_game = True
if states[2][0]==states[1][1]==states[0][2]!=0:
b[2][0].configure(bg='grey') b[1][1].configure(bg='grey')
b[0][2].configure(bg='grey') stop_game = True
root = Tk()
b = [[0,0,0], [0,0,0],

```


2 PYTHON BASIC FUNCTIONS

Parameters, Unpacking, Variables

In this exercise we will cover one more input method you can use to pass variables to a script (*script* being another name for your .py files). You know how you type `python3.6 ex13.py` to run the `ex13.py` file? Well the `ex13.py` part of the command is called an *argument*. What we'll do now is write a script that also accepts arguments.

Type this program and I'll explain it in detail:
`ex13.py`

```
from sys import argv
# read the WYSS section for how to run this
script, first, second, third = argv
4
print("The script is called:", script)
print("Your first variable is:", first)
print("Your second variable is:", second)
print("Your third variable is:", third)
```

On line 1 we have what's called an *import*. This is how you add features to your script from the Python feature set. Rather than give you all the features at once, Python asks you to say what you plan to use. This keeps your programs small, but it also acts as documentation for other programmers who read your code later.

The `argv` is the *argument variable*, a very standard name in programming that you will find used in many other languages. This variable *holds* the arguments you pass to your Python script when you run it. In the exercises you will get to play with this more and see what happens.

Line 3 *unpacks* `argv` so that, rather than holding all the arguments, it gets assigned to four variables you can work with: `script`, `first`, `second`, and `third`. This may look strange, but “unpack” is probably the best word to describe

what it does. It just says, “Take whatever is in argv, unpack it, and assign it to all of these variables on the left in order.”

After that we just print them out like normal.

Hold Up! Features Have Another Name

I call them “features” here (these little things you import to make your Python program do more) but nobody else calls them features. I just used that name because I needed to trick you into learning what they are without jargon. Before you can continue, you need to learn their real name: *modules*.

PARAMETERS, UNPACKING, VARIABLES

From now on we will be calling these “features” that we import *modules*. I’ll say things like, “You want to import the sys module.” They are also called “libraries” by other programmers, but let’s just stick with modules.

What You Should See

WARNING! Pay attention! You have been running python scripts without command line arguments. If you type only `python3.6 ex13.py` you are *doing it wrong!* Pay close attention to how I run it. This applies any time you see `argv` being used.

Run the program like this (and you *must* pass *three* command line arguments):

Exercise 13 Session

```
$ python3.6 ex13.py first 2nd 3rd
```

The script is called: `ex13.py`

Your first variable is: `first`

Your second variable is: `2nd` Your third variable is: `3rd`

This is what you should see when you do a few different runs with different arguments:

Exercise 13 Session

```
$ python3.6 ex13.py stuff things that The script is called: ex13.py
```

Your first variable is: stuff
Your second variable is: things
Your third variable is: that

\$

\$ python3.6 ex13.py apple orange grapefruit The script is called: ex13.py
Your first variable is: apple
Your second variable is: orange
Your third variable is: grapefruit

You can actually replace first, 2nd, and 3rd with any three things you want.
If you do not run it correctly, then you will get an error like this: Exercise 13 Session

\$ python3.6 ex13.py first 2nd Traceback (most recent call last): File "ex13.py", line 3, in <module> script, first, second, third = argv ValueError: not enough values to unpack (expected 4, got 3)

This happens when you do not put enough arguments on the command when you run it (in this case just first 2nd). Notice that when I run it I give it first 2nd, which caused it to give an error about “need more than 3 values to unpack” telling you that you didn’t give it enough parameters.

EXERCISE 14 Prompting and Passing

L

et’s do one exercise that uses argv and input together to ask the user something specific. You will need this for the next exercise where you learn to read and write files. In this exercise we’ll use input slightly differently by having it print a simple > prompt. This is similar to games like *Zork* or *Adventure*. ex14.py

```
1 from sys import argv
2
script, user_name = argv
prompt = '> '
5
print(f"Hi {user_name}, I'm the {script} script.") print("I'd like to ask you a
few questions.") print(f"Do you like me {user_name}?") likes =
```

```

input(prompt)
10
print(f"Where do you live {user_name}?" ) lives = input(prompt)
13
print("What kind of computer do you have?")
computer = input(prompt)
16
print(f"""

```

Alright, so you said {likes} about liking me. 19 You live in {lives}. Not sure where that is.

20 And you have a {computer} computer. Nice. 21 """)

We make a variable, prompt, that is set to the prompt we want, and we give that to input instead of typing it over and over. Now if we want to make the prompt something else, we just change it in this one spot and rerun the script. Very handy.

What You Should See

When you run this, remember that you have to give the script your name for the argv arguments.

Exercise 14 Session

```
$ python3.6 ex14.py zed
```

```
Hi zed, I'm the ex14.py script. I'd like to ask you a few questions. Do you like me zed? > Yes
```

PROMPTING AND PASSING

```
Where do you live zed?
```

```
> San Francisco
```

```
What kind of computer do you have? > Tandy 1000
```

```
Alright, so you said Yes about liking me.
```

You live in San Francisco. Not sure where that is. And you have a Tandy 1000 computer. Nice. Study Drills

Find out what the games *Zork* and *Adventure* were. Try to find copies and play them.

Change the prompt variable to something else entirely.

Add another argument and use it in your script, the same way you did in the previous exercise with `first, second = ARGV`.

Make sure you understand how I combined a `"""` style multiline string with the `{}` format activator as the last print.

Common Student Questions

I get `SyntaxError: invalid syntax` when I run this script. Again, you have to run it right on the command line, not inside Python. If you type `python3.6`, and then try to type `python3.6 ex14.py Zed`, it will fail because you are running Python *inside* Python. Close your window and then just type `python3.6 ex14.py Zed`.

I don't understand what you mean by changing the prompt? See the variable `prompt = '> '`. Change that to have a different value. You know this; it's just a string and you've done 13 exercises making them, so take the time to figure it out.

I get the error `ValueError: need more than 1 value to unpack`. Remember when I said you need to look at the *What You Should See* section and replicate what I did? You need to do the same thing here, and focus on how I type the command in and why I have a command line argument.

How can I run this from IDLE? Don't use IDLE.

Can I use double-quotes for the prompt variable? You totally can. Go ahead and try that.

You have a Tandy computer? I did when I was little.

I get `NameError: name 'prompt' is not defined` when I run it. You either spelled the name of the prompt variable wrong or forgot that line. Go back and compare each line of code to mine, starting at the bottom of the script and working up to the top. Any time you see this error, it means you spelled something wrong or forgot to create the variable.

EXERCISE 15 Reading Files

Y

ou know how to get input from a user with `input` or `argv`. Now you will learn about reading from a file. You may have to play with this exercise the most

to understand what's going on, so do the exercise carefully and remember your checks. Working with files is an easy way to *erase your work* if you are not careful.

This exercise involves writing two files. One is the usual `ex15.py` file that you will run, but the *other* is named `ex15_sample.txt`. This second file isn't a script but a plain text file we'll be reading in our script. Here are the contents of that file:

This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

What we want to do is *open* that file in our script and print it out. However, we do not want to just *hard code* the name `ex15_sample.txt` into our script. "Hard coding" means putting some bit of information that should come from the user as a string directly in our source code. That's bad because we want it to load other files later. The solution is to use `argv` or `input` to ask the user what file to open instead of hard coding the file's name. `ex15.py`

```
1 from sys import argv
2
3 script, filename = argv
4
5 txt = open(filename)
6
7 print(f"Here's your file {filename}:")
8 print(txt.read())
9
10 print("Type the filename again:")
11 file_again = input("> ")
12
13 txt_again = open(file_again)
14
15 print(txt_again.read())
```

A few fancy things are going on in this file, so let's break it down real quickly:

Lines 1 –3 use `argv` to get a filename. Next we have line 5, where we use a new command, `open`. Right now, run `pydoc open` and read the instructions. Notice how, like your own scripts and `input`, it takes a parameter and returns a value you can set to your own variable. You just opened a file.

READING FILES

Line 7 prints a little message, but on line 8 we have something very new and exciting. We call a function on `txt` named `read`. What you get back from `open` is a file, and it also has commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters, just like with `open` and `input`. The difference is that `txt.read()` says, “Hey `txt`! Do your `read` command with no parameters!”

The remainder of the file is more of the same, but we’ll leave the analysis to you in the Study Drills.

What You Should See

WARNING! Pay attention! I said, *pay attention!* You have been running scripts with just the name of the script, but now that you are using `argv` you have to add arguments. Look at the very first line of the example below and you will see I do `python ex15.py ex15_sample.txt` to run it. See the extra argument `ex15_sample.txt` after the `ex15.py` script name. If you do not type that you will get an error, so *pay attention!*

I made a file called `ex15_sample.txt` and ran my script. Exercise 15 Session

```
$ python3.6 ex15.py ex15_sample.txt Here's your file ex15_sample.txt: This is stuff I typed into a file.
```

```
It is really cool stuff.
```

```
Lots and lots of fun to have in here.
```

```
Type the filename again:
```

```
> ex15_sample.txt
```

```
This is stuff I typed into a file. It is really cool stuff.
```

```
Lots and lots of fun to have in here. EXERCISE 16 Reading and Writing Files
```

```
I
```

If you did the Study Drills from the last exercise, you should have seen all sorts of commands (methods/ functions) you can give to files. Here's the list of commands I want you to remember:

close: Closes the file. Like File->Save... in your editor.

read: Reads the contents of the file. You can assign the result to a variable.

readline: Reads just one line of a text file.

truncate: Empties the file. Watch out if you care about the file. write('stuff'): Writes "stuff" to the file.

seek(0): Moves the read/write location to the beginning of the file.

One way to remember what each of these does is to think of a vinyl record, cassette tape, VHS tape, DVD, or CD player. In the early days of computers, data was stored on each of these kinds of media, so many of the file operations still resemble a storage system that is linear. Tape and DVD drives need to "seek" a specific spot, and then you can read or write at that spot. Today we have operating systems and storage media that blur the lines between random access memory and disk drives, but we still use the older idea of a linear tape with a read/write head that must be moved.

For now, these are the important commands you need to know. Some of them take parameters, but we do not really care about that. You only need to remember that write takes a parameter of a string you want to write to the file.

Let's use some of this to make a simple little text editor: ex16.py

```
1 from sys import argv
```

```
2
```

```
3 script, filename = argv
```

```
4
```

```
print(f"We're going to erase {filename}.")
```

```
print("If you don't want that, hit CTRL-C (^C).")
```

```
print("If you do want that, hit RETURN.")
```

```
8
```

```
9 input("?")
```

```
10
```



```

11 print("Opening the file...") 12 target = open(filename, 'w')
13
14 print("Truncating the file. Goodbye!")
15 target.truncate()
16
17 print("Now I'm going to ask you for three lines.")
READING AND WRITING FILES
18
19 line1 = input("line 1: ")
20 line2 = input("line 2: ")
21 line3 = input("line 3: ")
22
23 print("I'm going to write these to the file.")
24
25 target.write(line1)
26 target.write("\n")
27 target.write(line2)
28 target.write("\n")
29 target.write(line3)
30 target.write("\n")
31
32 print("And finally, we close it.") 33 target.close()

```

That's a large file, probably the largest you have typed in. So go slow, do your checks, and make it run. One trick is to get bits of it running at a time. Get lines 1–8 running, then five more, then a few more, until it's all done and running.

What You Should See

There are actually two things you will see. First, the output of your new script:

Exercise 16 Session

```
$ python3.6 ex16.py test.txt
```

We're going to erase test.txt.

If you don't want that, hit CTRL-C (^C).

If you do want that, hit RETURN.

?

Opening the file...

Truncating the file. Goodbye! Now I'm going to ask you for three lines. line 1: Mary had a little lamb line 2: Its fleece was white as snow line 3: It was also tasty I'm going to write these to the file. And finally, we close it. Now, open up the file you made (in my case, test.txt) in your editor and check it out. Neat, right?

Write a script similar to the last exercise that uses read and argv to read the file you just created.

There's too much repetition in this file. Use strings, formats, and escapes to print out line1, line2, and line3 with just one target.write() command instead of six.

Find out why we had to pass a 'w' as an extra parameter to open. Hint: open tries to be safe by making you explicitly say you want to write a file.

If you open the file with 'w' mode, then do you really need the target.truncate()? Read the documentation for Python's open function and see if that's true.

EXERCISE 17 More Files

Now let's do a few more things with files. We'll write a Python script to copy one file to another. It'll be very short but will give you ideas about other things you can do with files. ex17.py

```
from sys import argv
from os.path import exists
3
4 script, from_file, to_file = argv
5
6 print(f"Copying from {from_file} to {to_file}") 7 # we could do these two
on one line, how?
in_file = open(from_file)
indata = in_file.read()
```

```

11
12 print(f"The input file is {len(indata)} bytes long") 13 print(f"Does the
output file exist? {exists(to_file)}") print("Ready, hit RETURN to continue,
CTRL-C to abort.") input()
17
out_file = open(to_file, 'w')
out_file.write(indata)
20
21 print("Alright, all done.")

22
23 out_file.close() 24 in_file.close()

```

You should immediately notice that we import another handy command named `exists`. This returns `True` if a file exists, based on its name in a string as an argument. It returns `False` if not. We'll be using this function in the second half of this book to do lots of things, but right now you should see how you can import it.

Using `import` is a way to get tons of free code other better (well, usually) programmers have written so you do not have to write it.

What You Should See

Just like your other scripts, run this one with two arguments: the file to copy from and the file to copy it to. I'm going to use a simple test file named `test.txt`:

Exercise 17 Session

```

$ # first make a sample file
$ echo "This is a test file." > test.txt
$ # then look at it
$ cat test.txt This is a test file.
$ # now run our script on it
$ python3.6 ex17.py test.txt new_file.txt
Copying from test.txt to new_file.txt
The input file is 21 bytes long
Does the output file exist? False

```

Ready, hit RETURN to continue, CTRL-C to abort. Alright, all done.

It should work with any file. Try a bunch more and see what happens. Just be careful you do not blast an important file.

WARNING! Did you see the trick I did with echo to make a file and cat to show the file? You can learn how to do that in the Appendix, “Command Line Crash Course.”

No way you can make this one line! That ; depends ; on ; how ; you ; define ; one ; line ; of ; code.

Is it normal to feel like this exercise was really hard? Yes, it is totally normal. Programming may not “click” for you until maybe even Exercise 36, or it might not until you finish the book and then make something with Python. Everyone is different, so just keep going and keep reviewing exercises that you had trouble with until it clicks. Be patient.

What does the len() function do? It gets the length of the string that you pass to it then returns that as a number. Play with it.

When I try to make this script shorter I get an error when I close the files at the end. You probably did something like `indata = open(from_file).read()`, which means you don’t need to then do `in_file.close()` when you reach the end of the script. It should already be closed by Python once that one line runs.

I get a Syntax:EOL while scanning string literal error. You forgot to end a string properly with a quote. Go look at that line again.

EXERCISE 18 Names, Variables, Code, Functions

B

ig title, right? I am about to introduce you to *the function!* Dum dum dah! Every programmer will go on and on about functions and all the different ideas about how they work and what they do, but I

will give you the simplest explanation you can use right now. Functions do

three things:

They name pieces of code the way variables name strings and numbers. They take arguments the way your scripts take argv.

Using 1 and 2, they let you make your own “mini-scripts” or “tiny commands.”

You can create a function by using the word def in Python. I’m going to have you make four different functions that work like your scripts, and then I’ll show you how each one is related.

ex18.py

```
1 # this one is like your scripts with argv 2 def print_two(*args): arg1, arg2 =  
args  
print(f"arg1: {arg1}, arg2: {arg2}")  
5  
  
6 # ok, that *args is actually pointless, we can just do this 7 def  
print_two_again(arg1, arg2):  
  
8 print(f"arg1: {arg1}, arg2: {arg2}")  
9  
10 # this just takes one argument 11 def print_one(arg1):  
  
12 print(f"arg1: {arg1}")  
13  
14 # this one takes no arguments 15 def print_none():  
16 print("I got nothin'.")  
17  
18  
print_two("Zed", "Shaw")  
print_two_again("Zed", "Shaw")  
print_one("First!")  
print_none()
```

Let’s break down the first function, print_two, which is the most similar to what you already know from making scripts:

We tell Python we want to make a function using `def` for “define.” NAMES, VARIABLES, CODE, FUNCTIONS

On the same line as `def` we give the function a name. In this case we just called it “`print_two`,” but it could also be “`peanuts`.” It doesn’t matter, except that your function should have a short name that says what it does.

We tell it we want `*args` (asterisk args), which is a lot like your `argv` parameter but for functions. This *has* to go inside parentheses to work. We end this line with a `:` (colon) and start indenting.

After the colon, all the lines that are indented four spaces will become attached to `print_two`.

Our first indented line is one that unpacks the arguments, the same as with your scripts.

To demonstrate how it works we print these arguments out, just like we would in a script.

The problem with `print_two` is that it’s not the easiest way to make a function. In Python we can skip the whole unpacking arguments and just use the names we want right inside parentheses. That’s what `print_two_again` does.

After that you have an example of how you make a function that takes one argument in `print_one`.

Finally you have a function that has no arguments in `print_none`.

WARNING! This is very important. Do *not* get discouraged right now if this doesn’t quite make sense. We’re going to do a few exercises linking functions to your scripts and show you how to make more. For now, just keep thinking “mini-script” when I say “function” and keep playing with them.

What You Should See

If you run `ex18.py` you should see: Exercise 18 Session

```
$ python3.6 ex18.py arg1: Zed, arg2: Shaw arg1: Zed, arg2: Shaw arg1:  
First!  
I got nothin'.
```

Right away you can see how a function works. Notice that you used your functions the way you use things like `exists`, `open`, and other commands. In fact, I've been tricking you because in Python those commands are just functions. This means you can make your own commands and use them in your scripts, too.

EXERCISE 19

Functions and Variables F

Functions may have been a mind-blowing amount of information, but do not worry. Just keep doing these exercises and going through your checklist from the last exercise and you will eventually get it.

There is one tiny point that you might not have realized, which we'll reinforce right now. The variables in your function are not connected to the variables in your script. Here's an exercise to get you thinking about this:

ex19.py

```
1 def cheese_and_crackers(cheese_count, boxes_of_crackers): 2 print(f"You
have {cheese_count} cheeses!")

print (f"You have {boxes_of_crackers} boxes of crackers!") print("Man that's
enough for a party!")
print("Get a blanket.\n")
6
7
print("We can just give the function numbers directly:")
cheese_and_crackers(20, 30)
10
11
print("OR, we can use variables from our script:")
amount_of_cheese = 10
amount_of_crackers = 50
15
16 cheese_and_crackers(amount_of_cheese, amount_of_crackers) 17
18
print("We can even do math inside too:")
```

```
cheese_and_crackers(10 + 20, 5 + 6)
```

```
21
```

```
22
```

```
print("And we can combine the two, variables and math:")
```

```
cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

This shows all the different ways we're able to give our function `cheese_and_crackers` the values it needs to print them. We can give it straight numbers. We can give it variables. We can give it math. We can even combine math and variables.

In a way, the arguments to a function are kind of like our `=` character when we make a variable. In fact, if you can use `=` to name something, you can usually pass it to a function as an argument.

FUNCTIONS AND VARIABLES

What You Should See

You should study the output of this script and compare it with what you think you should get for each of the examples in the script.

Exercise 19 Session

```
$ python3.6 ex19.py
```

We can just give the function numbers directly:

You have 20 cheeses!

You have 30 boxes of crackers!

Man that's enough for a party! Get a blanket.

OR, we can use variables from our script:

You have 10 cheeses!

You have 50 boxes of crackers!

Man that's enough for a party! Get a blanket.

We can even do math inside too:

You have 30 cheeses!

You have 11 boxes of crackers!

Man that's enough for a party! Get a blanket.

And we can combine the two, variables and math:

You have 110 cheeses!

You have 1050 boxes of crackers!
Man that's enough for a party! Get a blanket.

Is there a way to analyze what this function is doing so I can understand it better? There are many different ways, but try putting an English comment above each line describing what the line does. Another trick is to read the code out loud. Yet another is to print the code out and draw on the paper with pictures and comments showing what's going on.

What if I want to ask the user for the numbers of cheese and crackers? You need to use `int()` to convert what you get from `input()`.

Does making the variable `amount_of_cheese` change the variable `cheese_count` in the function? No, those variables are separate and live outside the function. They are then passed to the function, and temporary versions are made just for the function's run. When the function exits, these temporary variables go away and everything keeps working. Keep going in the book, and this should become clearer.

Is it bad to have global variables (like `amount_of_cheese`) with the same name as function variables? Yes, since then you're not quite sure which one you're talking about. But sometimes necessity means you have to use the same name, or you might do it on accident. Just avoid it whenever you can.

Is there a limit to the number of arguments a function can have? It depends on the version of Python and the computer you're on, but it is fairly large. The practical limit though is about five arguments before the function becomes annoying to use.

Can you call a function within a function? Yes, you'll make a game that does this later in the book.

This page intentionally left blank

EXERCISE 20 Functions and Files

R



Remember your checklist for functions, then do this exercise paying close attention to how functions and files can work together to make useful stuff.

ex20.py

```

1 from sys import argv
2
3 script, input_file = argv
4
5 def print_all(f):
6     print(f.read())
7
8 def rewind(f):
9     f.seek(0)
10
11 def print_a_line(line_count, f):
12     print(line_count, f.readline())
13
14 current_file = open(input_file)
15
16 print("First let's print the whole file:\n")
17
18 print_all(current_file)
19
20 print("Now let's rewind, kind of like a tape.")
21
22 rewind(current_file)
23
24 print("Let's print three lines:")
25
26 current_line = 1
27 print_a_line(current_line, current_file) 28
29 current_line = current_line + 1
30 print_a_line(current_line, current_file) 31
32 current_line = current_line + 1
33 print_a_line(current_line, current_file)

```

Pay close attention to how we pass in the current line number each time we run `print_a_line`.

FUNCTIONS AND FILES

What You Should See

Exercise 20 Session

\$ python3.6 ex20.py test.txt First let's print the whole file:

This is line 1

This is line 2

This is line 3

Now let's rewind, kind of like a tape.

Let's print three lines:

This is line ¹ This is line ² This is line ³₄₅

Why does seek(0) not set the current_line to 0? First, the seek() function is dealing in *bytes*, not lines. The code seek(0) moves the file to the 0 byte (first byte) in the file. Second, current_line is just a variable and has no real connection to the file at all. We are manually incrementing it.

What is += ? You know how in English I can rewrite “it is” as “it’s”? Or I can rewrite “you are” as “you’re”? In English this is called a contraction, and this is kind of like a contraction for the two operations = and +. That means x = x + y is the same as x += y.

How does readline() know where each line is? Inside readline() is code that scans each byte of the file until it finds a \n character, then stops reading the file to return what it found so far. The file f is responsible for maintaining the current position in the file after each readline() call, so that it will keep reading each line.

Why are there empty lines between the lines in the file? The readline() function returns the \n that’s in the file at the end of that line. Add a end = "" at the end of your print function calls to avoid adding double \n to every line.

¹
₃ 4 . 5d.

EXERCISE 21 Functions Can Return Something

Y

ou have been using the = character to name variables and set them to numbers or strings. We’re now going to blow your mind again by showing

you how to use = and a new Python word, return, to set variables to be a *value from a function*. There will be one thing to pay close attention to, but first type this in: ex21.py

```
def add(a, b):
    print(f"ADDING {a} + {b}")
    return a + b
4
def subtract(a, b):
    print(f"SUBTRACTING {a} - {b}")
    return a - b
8
def multiply(a, b):
    print(f"MULTIPLYING {a} * {b}")
    return a * b
12
def divide(a, b):
    print(f"DIVIDING {a} / {b}")
    return a / b
16
17
18 print("Let's do some math with just functions!")
19
age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)
24
25 print(f"Age: {age}, Height: {height}, Weight: {weight}, IQ: {iq}")
26
27
# A puzzle for the extra credit, type it in anyway.
print("Here is a puzzle.")
30

31 what = add(age, subtract(height, multiply(weight, divide(iq, 2))))
32
33 print("That becomes: ", what, "Can you do it by hand?")
```

We are now doing our own math functions for add, subtract, multiply, and divide. The important thing to notice is the last line where we say return a + b (in add). What this does is the following:

Our function is called with two arguments: a and b.

We print out what our function is doing, in this case “ADDING.”

FUNCTIONS CAN RETURN SOMETHING

We tell Python to do something kind of backward: we return the addition of a + b. You might say this as, “I add a and b, then return them.” Python adds the two numbers. Then when the function ends, any line that runs it will be able to assign this a + b result to a variable.

As with many other things in this book, you should take this real slow, break it down, and try to trace what’s going on. To help there is extra credit to solve a puzzle and learn something cool.

What You Should See

Exercise 21 Session

```
$ python3.6 ex21.py
```

Let's do some math with just functions!

ADDING 30 + 5

SUBTRACTING 78 - 4

MULTIPLYING 90 * 2

DIVIDING 100 / 2

Age: 35, Height: 74, Weight: 180, IQ: 50.0 Here is a puzzle. DIVIDING 50.0 / 2

MULTIPLYING 180 * 25.0

SUBTRACTING 74 - 4500.0

ADDING 35 + -4426.0

76

What Do You Know So Far?

T

here won’t be any code in this exercise, so there’s no *What You Should See* or *Study Drills* sections.

In fact, this exercise is like one giant *Study Drills*. I’m going to have you do a review of what you have learned so far.

First, go back through every exercise you have done so far and write down every word and symbol (another name for “character”) that you have used. Make sure your list of symbols is complete.

Next to each word or symbol, write its name and what it does. If you can't find a name for a symbol in this book, then look for it online. If you do not know what a word or symbol does, then read about it again and try using it in some code.

You may run into a few things you can't find out or know, so just keep those on the list and be ready to look them up when you find them.

Once you have your list, spend a few days rewriting the list and doublechecking that it's correct. This may get boring, but push through and really nail it down.

Once you have memorized the list and what they do, then step it up by writing out tables of symbols, their names, and what they do *from memory*. If you hit some you can't recall from memory, go back and memorize them again.

WARNING! The most important thing when doing this exercise is, "There is no failure, only

trying."

What You Are Learning

It's important when you are doing a boring, mindless memorization exercise like this to know why. It helps you focus on a goal and know the purpose of all your efforts.

In this exercise you are learning the names of symbols so that you can read source code more easily. It's similar to learning the alphabet and basic words of English, except this Python alphabet has extra symbols you might not know.

Just take it slow and do not hurt your brain. It's best to take 15 minutes at a time with your list and then take a break. Giving your brain a rest will help you learn faster with less frustration.

Strings, Bytes, and Character Encodings

T

o do this exercise you'll need to *download* a text file that I've written named languages.txt ([https:// learnpythonthehardway.org/python3/languages.txt](https://learnpythonthehardway.org/python3/languages.txt)). This file was created with a list of human lan

guages to demonstrate a few interesting concepts:

How modern computers store human languages for display and processing and how Python 3 calls this strings

How you must “encode” and “decode” Python's strings into a type called bytes

How to handle errors in your string and byte handling

How to read code and find out what it means even if you've never seen it before

In addition to that you'll also get a brief glimpse of the Python 3 if-statement and lists for processing a list of things. You don't have to master this code or understand these concepts right away. You'll get plenty of practice in later exercises. For now your job is to get a taste of the future and learn the four topics in the preceding list.

WARNING! This exercise is hard! There's a lot of information in it that you need to understand, and it's information that goes deep into computers. This exercise is complex because Python's strings are complex and difficult to use. I recommend you take this exercise *painfully* slow. Write down every word you don't understand, and look it up or research it. Take a paragraph at a time if you must. You can continue with other exercises while you study this one, so don't get stuck here. Just chip away at it for as long as it takes.

Initial Research

I'm going to teach you how to research a piece of code to expose its secrets. You'll need the languages.txt file for this code to work, so make sure you download it first. The languages.txt file simply contains a list of human language names that are encoded in UTF-8. ex23.py

```
import sys
```

```
script, input_encoding, error = sys.argv 3 4
```

```
def main(language_file, encoding, errors): line = language_file.readline()
```

```
7
```

STRINGS, BYTES, AND CHARACTER ENCODINGS [if](#) line:

```
print_line(line, encoding, errors)
return main(language_file, encoding, errors)
11
12
def print_line(line, encoding, errors):
    next_lang = line.strip()
    raw_bytes = next_lang.encode(encoding, errors=errors)
    cooked_string = raw_bytes.decode(encoding, errors=errors)
17

18 print(raw_bytes, "<===>", cooked_string)
19
20
21 languages = open("languages.txt", encoding="utf-8")
22
23 main(languages, input_encoding, error)
```

Simply write down a list of each thing you've never seen before. There may be quite a few things that are new, so scan the file a few times. Once you have that you'll want to run this Python script to play with it. Here are some example commands I used to test it:



```
$ python3.6 ex23.py utf-8 strict
b'Afrikaans' <===> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <===> ჰაშაფი
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <===> Ançwəa
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <===> العربية
b'V\xc3\xb5ro' <===> Võro
b'\xe6\x96\x87\xe8\xa8\x80' <===> 文言
b'\xe5\x90\xb4\xe8\xaf\xad' <===> 吴语
b'\xd7\x99\xd7\x99\xd6\xb4\xd7\x93\xd7\x99\xd7\xa9' <===> 上海话
b'\xe4\xb8\xad\xe6\x96\x87' <===> 中文
$
```

WARNING! You'll notice I'm using images here to show you what you should see. After extensive testing it turns out that so many people have their computers configured to not display UTF8 that I had to use images so you'll know what to expect. Even my own typesetting system (LaTeX) couldn't handle these encodings, forcing me to use images instead. If you don't see this then your Terminal is most likely not able to display UTF-8 and you should try to fix that.

These examples use the utf-8, utf-16, and big5 encodings to demonstrate the conversion and the types of errors you can get. Each of these names are called a “codec” in Python 3, but you use the parameter “encoding.” At the end of this exercise there’s a list of the available encodings if you want to try more. I’ll cover what all of this output means shortly. You’re only trying to get an idea of how this works so we can talk about it.

After you’ve run it a few times, go through your list of symbols and make a guess as to what they do. When you’ve written down your guesses try looking the symbols up online to see if you can confirm your hypotheses. Don’t worry if you have no idea how to search for them. Just give it a try.

Switches, Conventions, and Encodings

Before I can get into what this code means, you need to learn some basics about how data is stored in a computer. Modern computers are incredibly complex, but at their cores they are like a huge array of light switches. Computers use electricity to flip switches on or off. These switches can represent 1 for on, or 0 for off. In the old days there were all kinds of weird computers that did more than just 1 or 0, but these days it’s just 1s and 0s. One represents energy, electricity, on, power, substance. Zero represents off, done, gone, power down, the lack of energy. We call these 1s and 0s “bits.”

Now, a computer that only lets you work with 1 and 0 would be both horribly inefficient and incredibly annoying. Computers take these 1s and 0s and use them to encode larger numbers. At the small end a computer will use 8 of these 1s and 0s to encode 256 numbers (0-255). What does “encode” mean, though? It’s nothing more than an agreedupon standard for how a sequence of bits should represent a number. It’s a convention humans picked or stumbled on that says that 00000000 would be 0, 11111111 would be 255, and 00001111 would be 15. There were even huge wars in the early history of computers on nothing more than the order of these bits because they were simply conventions we all had to agree on.

Today we call a “byte” a sequence of 8 bits (1s and 0s). In the old days everyone had their own convention for a byte, so you’ll still run into people who think that this term should be flexible and handle sequences of 9 bits, 7 bits, or 6 bits, but now we just say it’s 8 bits. That’s our convention, and that

convention defines our encoding for a byte. There are further conventions for encoding large numbers using 16, 32, 64, and even more bits if you get into really big math. There's entire standards groups who do nothing but argue about these conventions, then implement them as encodings that eventually turn switches on and off.

Once you have bytes you can start to store and display text by deciding on another convention for how a number maps to a letter. In the early days of computing there were many conventions that mapped 8 or 7 bits (or less or more) onto lists of characters kept inside a computer. The most popular convention ended up being American Standard Code for Information Interchange, or ASCII. This standard maps a number to a letter. The number 90 is Z, which in bits is 1011010, which gets mapped to the ASCII table inside the computer.

You can try this out in Python right now:

```
>>> 0b1011010
90
>>> ord('Z')
90
>>> chr(90) 'Z'
>>>
```

First, I write the number 90 in binary, then I get the number based on the letter 'Z', then I convert the number to the letter 'Z'. Don't worry about needing to remember this though. I think I've had to do it twice the entire time I've used Python.

Once we have the ASCII convention for encoding a character using 8 bits (a byte), we can then “string” them together to make a word. If I want to write my name, “Zed A. Shaw,” I just use a sequence of bytes: [90, 101, 100, 32, 65, 46, 32, 83, 104, 97, 119]. Most of the early text in computers was nothing more than sequences of bytes, stored in memory, that a computer used to display text to a person. Again, this is just a sequence of conventions that turned switches on and off.

The problem with ASCII is that it only encodes English and maybe a few other similar languages. Remember that a byte can hold 256 numbers (0-255, or 00000000-11111111). Turns out, there's a *lot* more characters than 256

used throughout the world's languages. Different countries created their own encoding conventions for their languages, and that mostly worked, but many encodings could only handle one language. That meant if you want to put the title of an American English book in the middle of a Thai sentence you were kind of in trouble. You'd need one encoding for Thai and one for English.

To solve this problem a group of people created Unicode. It sounds like "encode," and it is meant to be a "universal encoding" of all human languages. The solution Unicode provides is like the ASCII table, but it's huge by comparison. You can use 32 bits to encode a Unicode character, and that is more characters than we could possibly find. A 32-bit number means we can store 4,294,967,295 characters (2^{32}), which is enough space for every possible human language and probably a lot of alien ones too. Right now we use the extra space for important things like poop and smile emojis.

We now have a convention for encoding any characters we want, but 32 bits is 4 bytes ($32/8 == 4$) which means there is so much wasted space in most text we want to encode. We can also use 16 bits (2 bytes), but still there's going to be wasted space in most text. The solution is to use a clever convention to encode most common characters using 8 bits, and then "escape" into larger numbers when we need to encode more characters. That means we have one more convention that is nothing more than a compression encoding, making it possible for most common characters to use 8 bits and then escape out into 16 or 32 bits as needed.

The convention for encoding text in Python is called "UTF -8", which means "Unicode Transformation Format 8 Bits." It is a convention for encoding Unicode characters into sequences of bytes, which are sequences of bits, which turn sequences of switches on and off. You can also use other conventions (encodings), but UTF-8 is the current standard.

Disecting the Output

We can now look at the output of the commands shown previously. Let's take just that first command and the first few lines of output:

```
python — bash — 80x24
$ python3.6 ex23.py utf-8 strict
b'Afrikaans' <====> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <====> አማርኛ
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <====> Англи
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <====> العربية
b'\xc3\xb5ro' <====> Vöro
b'\xe6\x96\x87\xe8\xa8\x80' <====> 文言
b'\xe5\x90\xb4\xe8\xaf\xad' <====> 吴语
b'\xd7\x99\xd7\x99\xd6\xb4\xd7\x93\xd7\x99\xd7\xa9' <====> 𐤆𐤒𐤕
b'\xe4\xb8\xad\xe6\x96\x87' <====> 中文
$
```

The ex23.py script is taking bytes written inside the b" (byte string) and converting them to the UTF-8 (or other) encoding you specified. On the left is the numbers for each byte of the UTF-8 (shown in hexadecimal), and the right has the character output as actual UTF-8. The way to think of this is the left side of <====> is the Python numerical bytes, or the “raw” bytes Python uses to store the string. You specify this with b" to tell Python this is bytes. These raw bytes are then displayed “cooked” on the right so you can see the real characters in your Terminal.

Disecting the Code

We have an understanding of strings and byte sequences. In Python a string is a UTF-8 encoded sequence of characters for displaying or working with text. The bytes are then the “raw” sequence of bytes that Python uses to store this UTF-8 string and start with a b' to tell Python you are working with raw bytes. This is all based on conventions for how Python wants to work with text. Here’s a Python session showing me encoding strings and decoding bytes.

```
python — bash — 82x34
$ python3.6
Python 3.6.0 (default, Feb  2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> raw_bytes = b'\xe6\x96\x87\xe8\xa8\x80'
>>> utf_string = "文言"
>>> raw_bytes.decode()
'文言'
>>> utf_string.encode()
b'\xe6\x96\x87\xe8\xa8\x80'
>>> raw_bytes == utf_string.encode()
True
>>> utf_string == raw_bytes.decode()
True
>>>
>>> quit()
$
```

All you need to remember is if you have raw bytes, then you must use .decode() to get the string. Raw bytes have no convention to them. They are

just sequences of bytes with no meaning other than numbers, so you must tell Python to “decode this into a UTF string.” If you have a string and want to send it, store it, share it, or do some other operation, then usually it’ll work, but sometimes Python will throw up an error saying it doesn’t know how to “encode” it. Again, Python knows its internal convention, but it has no idea what convention you need. In that case, you must use `.encode()` to get the bytes you need.

The way to remember this (even though I look it up almost every time) is to remember the mnemonic DBES, which stands for *Decode Bytes, Encode Strings*. I say “deebess” in my head when I have to convert bytes and strings. When you have bytes and need a string, decode bytes. When you have a string and need bytes, encode strings.

With that in mind, let’s break down the code in `ex23.py` line-by-line: 1–2 I start with the usual command line argument handling that you already know.

I start the main meat of this code in a function conveniently called `main`. This will be called at the end of this script to get things going.

The first thing this function does is read one line from the `languages` file it is given. You have done this before so nothing new here. Just `readline` as before when dealing with text files.

I use something new. You will learn about this in the second half of the book, so consider this a teaser of interesting things to come. This is an `if`-statement, and it lets you make decisions in your Python code. You can test the truth of a variable and, based on that truth, run a piece of code or not run it. In this case I’m testing whether `line` has something in it. The `readline` function will return an empty string when it reaches the end of the file and `line` simply tests for this empty string. As long as `readline` gives us something, this will be true, and the code *under* (indented in, lines 9–10) will run. When this is false, Python will skip lines 9–10.

I call a separate function to do the actual printing of this line. This simplifies my code and makes it easier for me to understand it. If I want to learn what this function does, I can jump to it and study. Once I know what `print_line` does I can attach my memory to the name `print_line` and forget about the

details.

I have written a tiny yet powerful piece of magic here. I am calling main again inside main. Actually it's not magic since nothing really is magical in programming. All the information you need is there. This looks like I am calling the function *inside* itself, which seems like it should be illegal to do. Ask yourself, why should that be illegal? There's no technical reason why I can't call any function I want right there, even this main function. If a function is simply a jump to the top where I've named it main, then calling this function at the end of itself would . . . jump back to the top and run it again. That would make it loop. Now look back at line 8, and you'll

see the if-statement keeps this function from looping forever. Carefully study this because it is a significant concept, but don't worry if you can't grasp it right away.

I start the definition for the print_line function, which does the actual encoding of each line from the languages.txt file.

This is a simple stripping of the trailing \n on the line string.

I *finally* take the language I've received from the languages.txt file and encode it into the raw bytes. Remember the DBES mnemonic: Decode bytes, encode strings. The next_lang variable is a string, so to get the raw bytes I must call .encode() on it to encode strings. I pass to encode() the encoding I want and how to handle errors.

I do the extra step of showing the inverse of line 15 by creating a cooked_string variable from the raw_bytes. Remember, DBES says I decode bytes, and raw_bytes is bytes, so I call .decode() on it to get a Python string. This string should be the same as the next_lang variable.

18 I simply print them both out to show you what they look like. 21 I'm done defining functions, so now I want to open the languages.txt file.

23 The end of the script simply runs the main function with all the correct parameters to get everything going and kick-start the loop. Remember that this then jumps to where the main function is defined on line 5, and on line 10 main is called again, causing this to keep looping.

The if line: on line 8 will prevent our loop from going forever. Encodings Deep Dive

We can now use our little script to explore other encodings. The following shows me playing with different encodings and seeing how to break them. First, I'm doing a simple UTF-16 encoding so you can see how it changes compared to UTF-8. You can also use UTF-32 to see how that's even bigger and get an idea of the space saved with UTF-8. After that I try Big5, and you'll see that Python does *not* like that at all. It throws up an error that Big5 can't encode some of the characters at position 0 (which is super helpful). One solution is to tell Python to "replace" any bad characters for the Big5 encoding. That's the next example, and you'll see it puts a ? character wherever it finds a character that doesn't match the Big5 encoding system.

```
python -- bash -- 82x34
$ python3.6 ex23.py utf-16 strict
b'\xff\xfeA\x0f\x00r\x00i\x00k\x00a\x00a\x00n\x00s\x00' <====> Afrikaans
b'\xff\xfe\x12\x1b\x12-\x12\x9b\x12' <====> 𐀀𐀁𐀂
b'\xff\xfe\x10\x04\xa7\x04A\x04H\x04\xd9\x040\x04' <====> Angola
b'\xff\xfe'\x060'\x069'\x061'\x06(\x06J'\x06)\x06" <====> العربية
b'\xff\xfeV\x00\x0f5\x00r\x00o\x00' <====> Vöro
b'\xff\xfe\x87e\x00\x8a' <====> 文青
b'\xff\xfe4T\xed\x8b' <====> 英語
b'\xff\xfe\x05\x09\x05\x05\x04\x05\x03\x05\x09\x05\x05' <====> 𐀀𐀁𐀂𐀃𐀄𐀅𐀆𐀇𐀈𐀉𐀊𐀋𐀌𐀍𐀎𐀏𐀐𐀑𐀒𐀓𐀔𐀕𐀖𐀗𐀘𐀙𐀚𐀛𐀜𐀝𐀞𐀟𐀠𐀡𐀢𐀣𐀤𐀥𐀦𐀧𐀨𐀩𐀪𐀫𐀬𐀭𐀮𐀯𐀰𐀱𐀲𐀳𐀴𐀵𐀶𐀷𐀸𐀹𐀺𐀻𐀼𐀽𐀾𐀿𐁀𐁁𐁂𐁃𐁄𐁅𐁆𐁇𐁈𐁉𐁊𐁋𐁌𐁍𐁎𐁏𐁐𐁑𐁒𐁓𐁔𐁕𐁖𐁗𐁘𐁙𐁚𐁛𐁜𐁝𐁞𐁟𐁠𐁡𐁢𐁣𐁤𐁥𐁦𐁧𐁨𐁩𐁪𐁫𐁬𐁭𐁮𐁯𐁰𐁱𐁲𐁳𐁴𐁵𐁶𐁷𐁸𐁹𐁺𐁻𐁼𐁽𐁾𐁿𐂀𐂁𐂂𐂃𐂄𐂅𐂆𐂇𐂈𐂉𐂊𐂋𐂌𐂍𐂎𐂏𐂐𐂑𐂒𐂓𐂔𐂕𐂖𐂗𐂘𐂙𐂚𐂛𐂜𐂝𐂞𐂟𐂠𐂡𐂢𐂣𐂤𐂥𐂦𐂧𐂨𐂩𐂪𐂫𐂬𐂭𐂮𐂯𐂰𐂱𐂲𐂳𐂴𐂵𐂶𐂷𐂸𐂹𐂺𐂻𐂼𐂽𐂾𐂿𐃀𐃁𐃂𐃃𐃄𐃅𐃆𐃇𐃈𐃉𐃊𐃋𐃌𐃍𐃎𐃏𐃐𐃑𐃒𐃓𐃔𐃕𐃖𐃗𐃘𐃙𐃚𐃛𐃜𐃝𐃞𐃟𐃠𐃡𐃢𐃣𐃤𐃥𐃦𐃧𐃨𐃩𐃪𐃫𐃬𐃭𐃮𐃯𐃰𐃱𐃲𐃳𐃴𐃵𐃶𐃷𐃸𐃹𐃺𐃻𐃼𐃽𐃾𐃿𐄀𐄁𐄂𐄃𐄄𐄅𐄆𐄇𐄈𐄉𐄊𐄋𐄌𐄍𐄎𐄏𐄐𐄑𐄒𐄓𐄔𐄕𐄖𐄗𐄘𐄙𐄚𐄛𐄜𐄝𐄞𐄟𐄠𐄡𐄢𐄣𐄤𐄥𐄦𐄧𐄨𐄩𐄪𐄫𐄬𐄭𐄮𐄯𐄰𐄱𐄲𐄳𐄴𐄵𐄶𐄷𐄸𐄹𐄺𐄻𐄼𐄽𐄾𐄿𐅀𐅁𐅂𐅃𐅄𐅅𐅆𐅇𐅈𐅉𐅊𐅋𐅌𐅍𐅎𐅏𐅐𐅑𐅒𐅓𐅔𐅕𐅖𐅗𐅘𐅙𐅚𐅛𐅜𐅝𐅞𐅟𐅠𐅡𐅢𐅣𐅤𐅥𐅦𐅧𐅨𐅩𐅪𐅫𐅬𐅭𐅮𐅯𐅰𐅱𐅲𐅳𐅴𐅵𐅶𐅷𐅸𐅹𐅺𐅻𐅼𐅽𐅾𐅿𐆀𐆁𐆂𐆃𐆄𐆅𐆆𐆇𐆈𐆉𐆊𐆋𐆌𐆍𐆎𐆏𐆐𐆑𐆒𐆓𐆔𐆕𐆖𐆗𐆘𐆙𐆚𐆛𐆜𐆝𐆞𐆟𐆠𐆡𐆢𐆣𐆤𐆥𐆦𐆧𐆨𐆩𐆪𐆫𐆬𐆭𐆮𐆯𐆰𐆱𐆲𐆳𐆴𐆵𐆶𐆷𐆸𐆹𐆺𐆻𐆼𐆽𐆾𐆿𐇀𐇁𐇂𐇃𐇄𐇅𐇆𐇇𐇈𐇉𐇊𐇋𐇌𐇍𐇎𐇏𐇐𐇑𐇒𐇓𐇔𐇕𐇖𐇗𐇘𐇙𐇚𐇛𐇜𐇝𐇞𐇟𐇠𐇡𐇢𐇣𐇤𐇥𐇦𐇧𐇨𐇩𐇪𐇫𐇬𐇭𐇮𐇯𐇰𐇱𐇲𐇳𐇴𐇵𐇶𐇷𐇸𐇹𐇺𐇻𐇼𐇽𐇾𐇿𐈀𐈁𐈂𐈃𐈄𐈅𐈆𐈇𐈈𐈉𐈊𐈋𐈌𐈍𐈎𐈏𐈐𐈑𐈒𐈓𐈔𐈕𐈖𐈗𐈘𐈙𐈚𐈛𐈜𐈝𐈞𐈟𐈠𐈡𐈢𐈣𐈤𐈥𐈦𐈧𐈨𐈩𐈪𐈫𐈬𐈭𐈮𐈯𐈰𐈱𐈲𐈳𐈴𐈵𐈶𐈷𐈸𐈹𐈺𐈻𐈼𐈽𐈾𐈿𐉀𐉁𐉂𐉃𐉄𐉅𐉆𐉇𐉈𐉉𐉊𐉋𐉌𐉍𐉎𐉏𐉐𐉑𐉒𐉓𐉔𐉕𐉖𐉗𐉘𐉙𐉚𐉛𐉜𐉝𐉞𐉟𐉠𐉡𐉢𐉣𐉤𐉥𐉦𐉧𐉨𐉩𐉪𐉫𐉬𐉭𐉮𐉯𐉰𐉱𐉲𐉳𐉴𐉵𐉶𐉷𐉸𐉹𐉺𐉻𐉼𐉽𐉾𐉿𐊀𐊁𐊂𐊃𐊄𐊅𐊆𐊇𐊈𐊉𐊊𐊋𐊌𐊍𐊎𐊏𐊐𐊑𐊒𐊓𐊔𐊕𐊖𐊗𐊘𐊙𐊚𐊛𐊜𐊝𐊞𐊟𐊠𐊡𐊢𐊣𐊤𐊥𐊦𐊧𐊨𐊩𐊪𐊫𐊬𐊭𐊮𐊯𐊰𐊱𐊲𐊳𐊴𐊵𐊶𐊷𐊸𐊹𐊺𐊻𐊼𐊽𐊾𐊿𐋀𐋁𐋂𐋃𐋄𐋅𐋆𐋇𐋈𐋉𐋊𐋋𐋌𐋍𐋎𐋏𐋐𐋑𐋒𐋓𐋔𐋕𐋖𐋗𐋘𐋙𐋚𐋛𐋜𐋝𐋞𐋟𐋠𐋡𐋢𐋣𐋤𐋥𐋦𐋧𐋨𐋩𐋪𐋫𐋬𐋭𐋮𐋯𐋰𐋱𐋲𐋳𐋴𐋵𐋶𐋷𐋸𐋹𐋺𐋻𐋼𐋽𐋾𐋿𐌀𐌁𐌂𐌃𐌄𐌅𐌆𐌇𐌈𐌉𐌊𐌋𐌌𐌍𐌎𐌏𐌐𐌑𐌒𐌓𐌔𐌕𐌖𐌗𐌘𐌙𐌚𐌛𐌜𐌝𐌞𐌟𐌠𐌡𐌢𐌣𐌤𐌥𐌦𐌧𐌨𐌩𐌪𐌫𐌬𐌭𐌮𐌯𐌰𐌱𐌲𐌳𐌴𐌵𐌶𐌷𐌸𐌹𐌺𐌻𐌼𐌽𐌾𐌿𐍀𐍁𐍂𐍃𐍄𐍅𐍆𐍇𐍈𐍉𐍊𐍋𐍌𐍍𐍎𐍏𐍐𐍑𐍒𐍓𐍔𐍕𐍖𐍗𐍘𐍙𐍚𐍛𐍜𐍝𐍞𐍟𐍠𐍡𐍢𐍣𐍤𐍥𐍦𐍧𐍨𐍩𐍪𐍫𐍬𐍭𐍮𐍯𐍰𐍱𐍲𐍳𐍴𐍵𐍶𐍷𐍸𐍹𐍺𐍻𐍼𐍽𐍾𐍿𐎀𐎁𐎂𐎃𐎄𐎅𐎆𐎇𐎈𐎉𐎊𐎋𐎌𐎍𐎎𐎏𐎐𐎑𐎒𐎓𐎔𐎕𐎖𐎗𐎘𐎙𐎚𐎛𐎜𐎝𐎞𐎟𐎠𐎡𐎢𐎣𐎤𐎥𐎦𐎧𐎨𐎩𐎪𐎫𐎬𐎭𐎮𐎯𐎰𐎱𐎲𐎳𐎴𐎵𐎶𐎷𐎸𐎹𐎺𐎻𐎼𐎽𐎾𐎿𐏀𐏁𐏂𐏃𐏄𐏅𐏆𐏇𐏈𐏉𐏊𐏋𐏌𐏍𐏎𐏏𐏐𐏑𐏒𐏓𐏔𐏕𐏖𐏗𐏘𐏙𐏚𐏛𐏜𐏝𐏞𐏟𐏠𐏡𐏢𐏣𐏤𐏥𐏦𐏧𐏨𐏩𐏪𐏫𐏬𐏭𐏮𐏯𐏰𐏱𐏲𐏳𐏴𐏵𐏶𐏷𐏸𐏹𐏺𐏻𐏼𐏽𐏾𐏿𐐀𐐁𐐂𐐃𐐄𐐅𐐆𐐇𐐈𐐉𐐊𐐋𐐌𐐍𐐎𐐏𐐐𐐑𐐒𐐓𐐔𐐕𐐖𐐗𐐘𐐙𐐚𐐛𐐜𐐝𐐞𐐟𐐠𐐡𐐢𐐣𐐤𐐥𐐦𐐧𐐨𐐩𐐪𐐫𐐬𐐭𐐮𐐯𐐰𐐱𐐲𐐳𐐴𐐵𐐶𐐷𐐸𐐹𐐺𐐻𐐼𐐽𐐾𐐿𐑀𐑁𐑂𐑃𐑄𐑅𐑆𐑇𐑈𐑉𐑊𐑋𐑌𐑍𐑎𐑏𐑐𐑑𐑒𐑓𐑔𐑕𐑖𐑗𐑘𐑙𐑚𐑛𐑜𐑝𐑞𐑟𐑠𐑡𐑢𐑣𐑤𐑥𐑦𐑧𐑨𐑩𐑪𐑫𐑬𐑭𐑮𐑯𐑰𐑱𐑲𐑳𐑴𐑵𐑶𐑷𐑸𐑹𐑺𐑻𐑼𐑽𐑾𐑿𐒀𐒁𐒂𐒃𐒄𐒅𐒆𐒇𐒈𐒉𐒊𐒋𐒌𐒍𐒎𐒏𐒐𐒑𐒒𐒓𐒔𐒕𐒖𐒗𐒘𐒙𐒚𐒛𐒜𐒝𐒞𐒟𐒠𐒡𐒢𐒣𐒤𐒥𐒦𐒧𐒨𐒩𐒪𐒫𐒬𐒭𐒮𐒯𐒰𐒱𐒲𐒳𐒴𐒵𐒶𐒷𐒸𐒹𐒺𐒻𐒼𐒽𐒾𐒿𐓀𐓁𐓂𐓃𐓄𐓅𐓆𐓇𐓈𐓉𐓊𐓋𐓌𐓍𐓎𐓏𐓐𐓑𐓒𐓓𐓔𐓕𐓖𐓗𐓘𐓙𐓚𐓛𐓜𐓝𐓞𐓟𐓠𐓡𐓢𐓣𐓤𐓥𐓦𐓧𐓨𐓩𐓪𐓫𐓬𐓭𐓮𐓯𐓰𐓱𐓲𐓳𐓴𐓵𐓶𐓷𐓸𐓹𐓺𐓻𐓼𐓽𐓾𐓿𐔀𐔁𐔂𐔃𐔄𐔅𐔆𐔇𐔈𐔉𐔊𐔋𐔌𐔍𐔎𐔏𐔐𐔑𐔒𐔓𐔔𐔕𐔖𐔗𐔘𐔙𐔚𐔛𐔜𐔝𐔞𐔟𐔠𐔡𐔢𐔣𐔤𐔥𐔦𐔧𐔨𐔩𐔪𐔫𐔬𐔭𐔮𐔯𐔰𐔱𐔲𐔳𐔴𐔵𐔶𐔷𐔸𐔹𐔺𐔻𐔼𐔽𐔾𐔿𐕀𐕁𐕂𐕃𐕄𐕅𐕆𐕇𐕈𐕉𐕊𐕋𐕌𐕍𐕎𐕏𐕐𐕑𐕒𐕓𐕔𐕕𐕖𐕗𐕘𐕙𐕚𐕛𐕜𐕝𐕞𐕟𐕠𐕡𐕢𐕣𐕤𐕥𐕦𐕧𐕨𐕩𐕪𐕫𐕬𐕭𐕮𐕯𐕰𐕱𐕲𐕳𐕴𐕵𐕶𐕷𐕸𐕹𐕺𐕻𐕼𐕽𐕾𐕿𐖀𐖁𐖂𐖃𐖄𐖅𐖆𐖇𐖈𐖉𐖊𐖋𐖌𐖍𐖎𐖏𐖐𐖑𐖒𐖓𐖔𐖕𐖖𐖗𐖘𐖙𐖚𐖛𐖜𐖝𐖞𐖟𐖠𐖡𐖢𐖣𐖤𐖥𐖦𐖧𐖨𐖩𐖪𐖫𐖬𐖭𐖮𐖯𐖰𐖱𐖲𐖳𐖴𐖵𐖶𐖷𐖸𐖹𐖺𐖻𐖼𐖽𐖾𐖿𐗀𐗁𐗂𐗃𐗄𐗅𐗆𐗇𐗈𐗉𐗊𐗋𐗌𐗍𐗎𐗏𐗐𐗑𐗒𐗓𐗔𐗕𐗖𐗗𐗘𐗙𐗚𐗛𐗜𐗝𐗞𐗟𐗠𐗡𐗢𐗣𐗤𐗥𐗦𐗧𐗨𐗩𐗪𐗫𐗬𐗭𐗮𐗯𐗰𐗱𐗲𐗳𐗴𐗵𐗶𐗷𐗸𐗹𐗺𐗻𐗼𐗽𐗾𐗿𐘀𐘁𐘂𐘃𐘄𐘅𐘆𐘇𐘈𐘉𐘊𐘋𐘌𐘍𐘎𐘏𐘐𐘑𐘒𐘓𐘔𐘕𐘖𐘗𐘘𐘙𐘚𐘛𐘜𐘝𐘞𐘟𐘠𐘡𐘢𐘣𐘤𐘥𐘦𐘧𐘨𐘩𐘪𐘫𐘬𐘭𐘮𐘯𐘰𐘱𐘲𐘳𐘴𐘵𐘶𐘷𐘸𐘹𐘺𐘻𐘼𐘽𐘾𐘿𐙀𐙁𐙂𐙃𐙄𐙅𐙆𐙇𐙈𐙉𐙊𐙋𐙌𐙍𐙎𐙏𐙐𐙑𐙒𐙓𐙔𐙕𐙖𐙗𐙘𐙙𐙚𐙛𐙜𐙝𐙞𐙟𐙠𐙡𐙢𐙣𐙤𐙥𐙦𐙧𐙨𐙩𐙪𐙫𐙬𐙭𐙮𐙯𐙰𐙱𐙲𐙳𐙴𐙵𐙶𐙷𐙸𐙹𐙺𐙻𐙼𐙽𐙾𐙿𐚀𐚁𐚂𐚃𐚄𐚅𐚆𐚇𐚈𐚉𐚊𐚋𐚌𐚍𐚎𐚏𐚐𐚑𐚒𐚓𐚔𐚕𐚖𐚗𐚘𐚙𐚚𐚛𐚜𐚝𐚞𐚟𐚠𐚡𐚢𐚣𐚤𐚥𐚦𐚧𐚨𐚩𐚪𐚫𐚬𐚭𐚮𐚯𐚰𐚱𐚲𐚳𐚴𐚵𐚶𐚷𐚸𐚹𐚺𐚻𐚼𐚽𐚾𐚿𐛀𐛁𐛂𐛃𐛄𐛅𐛆𐛇𐛈𐛉𐛊𐛋𐛌𐛍𐛎𐛏𐛐𐛑𐛒𐛓𐛔𐛕𐛖𐛗𐛘𐛙𐛚𐛛𐛜𐛝𐛞𐛟𐛠𐛡𐛢𐛣𐛤𐛥𐛦𐛧𐛨𐛩𐛪𐛫𐛬𐛭𐛮𐛯𐛰𐛱𐛲𐛳𐛴𐛵𐛶𐛷𐛸𐛹𐛺𐛻𐛼𐛽𐛾𐛿𐜀𐜁𐜂𐜃𐜄𐜅𐜆𐜇𐜈𐜉𐜊𐜋𐜌𐜍𐜎𐜏𐜐𐜑𐜒𐜓𐜔𐜕𐜖𐜗𐜘𐜙𐜚𐜛𐜜𐜝𐜞𐜟𐜠𐜡𐜢𐜣𐜤𐜥𐜦𐜧𐜨𐜩𐜪𐜫𐜬𐜭𐜮𐜯𐜰𐜱𐜲𐜳𐜴𐜵𐜶𐜷𐜸𐜹𐜺𐜻𐜼𐜽𐜾𐜿𐝀𐝁𐝂𐝃𐝄𐝅𐝆𐝇𐝈𐝉𐝊𐝋𐝌𐝍𐝎𐝏𐝐𐝑𐝒𐝓𐝔𐝕𐝖𐝗𐝘𐝙𐝚𐝛𐝜𐝝𐝞𐝟𐝠𐝡𐝢𐝣𐝤𐝥𐝦𐝧𐝨𐝩𐝪𐝫𐝬𐝭𐝮𐝯𐝰𐝱𐝲𐝳𐝴𐝵𐝶𐝷𐝸𐝹𐝺𐝻𐝼𐝽𐝾𐝿𐞀𐞁𐞂𐞃𐞄𐞅𐞆𐞇𐞈𐞉𐞊𐞋𐞌𐞍𐞎𐞏𐞐𐞑𐞒𐞓𐞔𐞕𐞖𐞗𐞘𐞙𐞚𐞛𐞜𐞝𐞞𐞟𐞠𐞡𐞢𐞣𐞤𐞥𐞦𐞧𐞨𐞩𐞪𐞫𐞬𐞭𐞮𐞯𐞰𐞱𐞲𐞳𐞴𐞵𐞶𐞷𐞸𐞹𐞺𐞻𐞼𐞽𐞾𐞿𐟀𐟁𐟂𐟃𐟄𐟅𐟆𐟇𐟈𐟉𐟊𐟋𐟌𐟍𐟎𐟏𐟐𐟑𐟒𐟓𐟔𐟕𐟖𐟗𐟘𐟙𐟚𐟛𐟜𐟝𐟞𐟟𐟠𐟡𐟢𐟣𐟤𐟥𐟦𐟧𐟨𐟩𐟪𐟫𐟬𐟭𐟮𐟯𐟰𐟱𐟲𐟳𐟴𐟵𐟶𐟷𐟸𐟹𐟺𐟻𐟼𐟽𐟾𐟿𐠀𐠁𐠂𐠃𐠄𐠅𐠆𐠇𐠈𐠉𐠊𐠋𐠌𐠍𐠎𐠏𐠐𐠑𐠒𐠓𐠔𐠕𐠖𐠗𐠘𐠙𐠚𐠛𐠜𐠝𐠞𐠟𐠠𐠡𐠢𐠣𐠤𐠥𐠦𐠧𐠨𐠩𐠪𐠫𐠬𐠭𐠮𐠯𐠰𐠱𐠲𐠳𐠴𐠵𐠶𐠷𐠸𐠹𐠺𐠻𐠼𐠽𐠾𐠿𐡀𐡁𐡂𐡃𐡄𐡅𐡆𐡇𐡈𐡉𐡊𐡋𐡌𐡍𐡎𐡏𐡐𐡑𐡒𐡓𐡔𐡕𐡖𐡗𐡘𐡙𐡚𐡛𐡜𐡝𐡞𐡟𐡠𐡡𐡢𐡣𐡤𐡥𐡦𐡧𐡨𐡩𐡪𐡫𐡬𐡭𐡮𐡯𐡰𐡱𐡲𐡳𐡴𐡵𐡶𐡷𐡸𐡹𐡺𐡻𐡼𐡽𐡾𐡿𐢀𐢁𐢂𐢃𐢄𐢅𐢆𐢇𐢈𐢉𐢊𐢋𐢌𐢍𐢎𐢏𐢐𐢑𐢒𐢓𐢔𐢕𐢖𐢗𐢘𐢙𐢚𐢛𐢜𐢝𐢞𐢟𐢠𐢡𐢢𐢣𐢤𐢥𐢦𐢧𐢨𐢩𐢪𐢫𐢬𐢭𐢮𐢯𐢰𐢱𐢲𐢳𐢴𐢵𐢶𐢷𐢸𐢹𐢺𐢻𐢼𐢽𐢾𐢿𐣀𐣁𐣂𐣃𐣄𐣅𐣆𐣇𐣈𐣉𐣊𐣋𐣌𐣍𐣎𐣏𐣐𐣑𐣒𐣓𐣔𐣕𐣖𐣗𐣘𐣙𐣚𐣛𐣜𐣝𐣞𐣟𐣠𐣡𐣢𐣣𐣤𐣥𐣦𐣧𐣨𐣩𐣪𐣫𐣬𐣭𐣮𐣯𐣰𐣱𐣲𐣳𐣴𐣵𐣶𐣷𐣸𐣹𐣺𐣻𐣼𐣽𐣾𐣿𐤀𐤁𐤂𐤃𐤄𐤅𐤆𐤇𐤈𐤉𐤊𐤋𐤌𐤍𐤎𐤏𐤐𐤑𐤒𐤓𐤔𐤕𐤖𐤗𐤘𐤙𐤚𐤛𐤜𐤝𐤞𐤟𐤠𐤡𐤢𐤣𐤤𐤥𐤦𐤧𐤨𐤩𐤪𐤫𐤬𐤭𐤮𐤯𐤰𐤱𐤲𐤳𐤴𐤵𐤶𐤷𐤸𐤹𐤺𐤻𐤼𐤽𐤾𐤿𐥀𐥁𐥂𐥃𐥄𐥅𐥆𐥇𐥈𐥉𐥊𐥋𐥌𐥍𐥎𐥏𐥐𐥑𐥒𐥓𐥔𐥕𐥖𐥗𐥘𐥙𐥚𐥛𐥜𐥝𐥞𐥟𐥠𐥡𐥢𐥣𐥤𐥥𐥦𐥧𐥨𐥩𐥪𐥫𐥬𐥭𐥮𐥯𐥰𐥱𐥲𐥳𐥴𐥵𐥶𐥷𐥸𐥹𐥺𐥻𐥼𐥽𐥾𐥿𐦀𐦁𐦂𐦃𐦄𐦅𐦆𐦇𐦈𐦉𐦊𐦋𐦌𐦍𐦎𐦏𐦐𐦑𐦒𐦓𐦔𐦕𐦖𐦗𐦘𐦙𐦚𐦛𐦜𐦝𐦞𐦟𐦠𐦡𐦢𐦣𐦤𐦥𐦦𐦧𐦨𐦩𐦪𐦫𐦬𐦭𐦮𐦯𐦰𐦱𐦲𐦳𐦴𐦵𐦶𐦷𐦸𐦹𐦺𐦻𐦼𐦽𐦾𐦿𐧀𐧁𐧂𐧃𐧄𐧅𐧆𐧇𐧈𐧉𐧊𐧋𐧌𐧍𐧎𐧏𐧐𐧑𐧒𐧓𐧔𐧕𐧖𐧗𐧘𐧙𐧚𐧛𐧜𐧝𐧞𐧟𐧠𐧡𐧢𐧣𐧤𐧥𐧦𐧧𐧨𐧩𐧪𐧫𐧬𐧭𐧮𐧯𐧰𐧱𐧲𐧳𐧴𐧵𐧶𐧷𐧸𐧹𐧺𐧻𐧼𐧽𐧾𐧿𐨀𐨁𐨂𐨃𐨄𐨅𐨆𐨇𐨈𐨉𐨊𐨋𐨌𐨍𐨎𐨏𐨐𐨑𐨒𐨓𐨔𐨕𐨖𐨗𐨘𐨙𐨚𐨛𐨜𐨝𐨞𐨟𐨠𐨡𐨢𐨣𐨤𐨥𐨦𐨧𐨨𐨩𐨪𐨫𐨬𐨭𐨮𐨯𐨰𐨱𐨲𐨳𐨴𐨵𐨶𐨷𐨹𐨺𐨸𐨻𐨼𐨽𐨾𐨿𐩀𐩁𐩂𐩃𐩄𐩅𐩆𐩇𐩈𐩉𐩊𐩋𐩌𐩍𐩎𐩏𐩐𐩑𐩒𐩓𐩔𐩕𐩖𐩗𐩘𐩙𐩚𐩛𐩜𐩝𐩞𐩟𐩠𐩡𐩢𐩣𐩤𐩥𐩦𐩧𐩨𐩩𐩪𐩫𐩬𐩭𐩮𐩯𐩰𐩱𐩲𐩳𐩴𐩵𐩶𐩷𐩸𐩹𐩺𐩻𐩼𐩽𐩾𐩿𐪀𐪁𐪂𐪃𐪄𐪅𐪆𐪇𐪈𐪉𐪊𐪋𐪌𐪍𐪎𐪏𐪐𐪑𐪒𐪓𐪔𐪕𐪖𐪗𐪘𐪙𐪚𐪛𐪜𐪝𐪞𐪟𐪠𐪡𐪢𐪣𐪤𐪥𐪦𐪧𐪨𐪩𐪪𐪫𐪬𐪭𐪮𐪯𐪰𐪱𐪲𐪳𐪴𐪵𐪶𐪷𐪸𐪹𐪺𐪻𐪼𐪽𐪾𐪿𐫀𐫁𐫂𐫃𐫄𐫅𐫆𐫇𐫈𐫉𐫊𐫋𐫌𐫍𐫎𐫏𐫐𐫑𐫒𐫓𐫔𐫕𐫖𐫗𐫘𐫙𐫚𐫛𐫜𐫝𐫞𐫟𐫠𐫡𐫢𐫣𐫤𐫦𐫥𐫧𐫨𐫩𐫪𐫫𐫬𐫭𐫮𐫯𐫰𐫱𐫲𐫳𐫴𐫵𐫶𐫷𐫸𐫹𐫺𐫻𐫼𐫽𐫾𐫿𐬀𐬁𐬂𐬃𐬄𐬅𐬆𐬇𐬈𐬉𐬊𐬋𐬌𐬍𐬎𐬏𐬐𐬑𐬒𐬓𐬔𐬕𐬖𐬗𐬘𐬙𐬚𐬛𐬜𐬝𐬞𐬟𐬠𐬡𐬢𐬣𐬤𐬥𐬦𐬧𐬨𐬩𐬪𐬫𐬬𐬭𐬮𐬯𐬰𐬱𐬲𐬳𐬴𐬵𐬶𐬷𐬸𐬹𐬺𐬻𐬼𐬽𐬾𐬿𐭀𐭁𐭂𐭃𐭄𐭅𐭆𐭇𐭈𐭉𐭊𐭋𐭌𐭍𐭎𐭏𐭐𐭑𐭒𐭓𐭔𐭕𐭖𐭗𐭘𐭙𐭚𐭛𐭜𐭝𐭞𐭟𐭠𐭡𐭢𐭣𐭤𐭥𐭦𐭧𐭨𐭩𐭪𐭫𐭬𐭭𐭮𐭯𐭰𐭱𐭲𐭳𐭴𐭵𐭶𐭷
```

effectively reversing the script.

If you can do that, then you can also *break* these bytes by removing some to see what happens. How much do you need to remove to cause Python to break? How much can you remove to damage the string output but pass Python's decoding system.

Use what you learned from item 4 to see if you can mangle the files. What errors do you get?

How much damage can you cause and get the file past Python's decoding system?

EXERCISE 24 More Practice

Y

ou are getting to the end of this section. You should have enough Python “under your fingers” to move on to learning about how programming really works, but you should do some more practice. This exercise is longer and all about building up stamina. The next exercise will be similar. Do them, get them exactly right, and do your checks.

ex24.py

```
print ("Let's practice everything.")
print('You\'d need to know \'bout escapes with \\ that do:') print('\n newlines
and \t tabs.')
```

4

```
poem = """
\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
```

```
nor comprehend passion from intuition 10 and requires an explanation 11
\n\t\twhere there is none.
```

12 """

13

```
print("-----")
```



```

print(poem)
print("-----")
17
18
five = 10 - 2 + 3 - 6
print(f"This should be five: {five}")
21
def secret_formula(started):
    jelly_beans = started * 500
    jars = jelly_beans / 1000
    crates = jars / 100
    return jelly_beans, jars, crates
27
28
start_point = 10000
beans, jars, crates = secret_formula(start_point) 31
# remember that this is another way to format a string
print("With a starting point of: {}".format(start_point))
# it's just like with an f"" string
print(f"We'd have {beans} beans, {jars} jars, and {crates} crates.") 36
37 start_point = start_point / 10
MORE PRACTICE
38
print("We can also do that this way:")
formula = secret_formula(start_point)
# this is an easy way to apply a list to a format string

print ("We'd have {} beans, {} jars, and {} crates.".format(*formula)) What
You Should See
Exercise 24 Session

```

\$ python3.6 ex24.py

Let's practice everything.

You'd need to know 'bout escapes with \ that do:

newlines and tabs.

The lovely world with logic so firmly planted
cannot discern
the needs of love
nor comprehend passion from intuition and requires an explanation where
there is none.

This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000.0 jars, and 50.0 crates. We can also do that
this way:
We'd have 500000.0 beans, 500.0 jars, and 5.0 crates. **XERCISE 25** Even
More Practice
W

e're going to do some more practice involving functions and variables to
make sure you know them well. This exercise should be straightforward for
you to type in, break down, and understand.

However, this exercise is a little different. You won't be running it. Instead
you will import it into Python and run the functions yourself. ex25.py

```
def break_words(stuff):  
    """This function will break up words for us.""" words = stuff.split(' ')  
    return words  
5  
def sort_words(words):  
    """Sorts the words."""  
    return sorted(words)  
9  
def print_first_word(words):  
    """Prints the first word after popping it off.""" word = words.pop(0)  
    print(word)  
14  
def print_last_word(words):  
    """Prints the last word after popping it off."""  
  
word = words.pop(-1)  
print(word)
```

19

```
def sort_sentence(sentence):  
    """Takes in a full sentence and returns the sorted words.""" words =  
    break_words(sentence)  
    return sort_words(words)
```

24

```
def print_first_and_last(sentence):  
    """Prints the first and last words of the sentence."""  
    words = break_words(sentence)  
    print_first_word(words)  
    print_last_word(words)
```

30

```
def print_first_and_last_sorted(sentence):  
    """Sorts the words then prints the first and last one."""  
    words = sort_sentence(sentence)  
    print_first_word(words)  
    print_last_word(words)
```

EVEN MORE PRACTICE 91

First, run this with `python3.6 ex25.py` to find any errors you have made. Once you have found all of the errors you can and fixed them, you will then want to follow the *What You Should See* section to complete the exercise.

What You Should See

In this exercise we're going to interact with your `ex25.py` file inside the `python3.6` interpreter you used periodically to do calculations. You run `python3.6` from the Terminal like this:

```
$ python3.6  
Python 3.6.0rc2 (v3.6.0rc2:800a67f7806d, Dec 16 2016, 14:12:21) [GCC  
4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin  
Type "help", "copyright", "credits" or "license" for more information. >>>
```

Your output should look like mine, and after the `>` character (called the prompt) you can type Python code in, and it will run immediately. Using this I want you to type each of these lines of Python code into `python3.6` and see what it does:

Exercise 25 Python Session

```
import ex25
sentence = "All good things come to those who wait." words =
ex25.break_words(sentence)
words
sorted_words = ex25.sort_words(words)
sorted_words
ex25.print_first_word(words)
ex25.print_last_word(words)
words
ex25.print_first_word(sorted_words)
ex25.print_last_word(sorted_words)
sorted_words
sorted_words = ex25.sort_sentence(sentence)
sorted_words
ex25.print_first_and_last(sentence)
ex25.print_first_and_last_sorted(sentence)
```

Here's what it looks like when I work with the ex25.py module in python3.6:

Exercise 25 Python Session

Python 3.6.0 (default, Feb 2 2017, 12:48:29)

[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import ex25
>>> sentence = "All good things come to those who wait." >>> words =
ex25.break_words(sentence)

>>> words
['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.'] >>> sorted_words =
ex25.sort_words(words) >>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who'] >>>
ex25.print_first_word(words)
All
>>> ex25.print_last_word(words) wait.
>>> words
['good', 'things', 'come', 'to', 'those', 'who']
```

```

>>> ex25.print_first_word(sorted_words)
All
>>> ex25.print_last_word(sorted_words) who
>>> sorted_words

['come', 'good', 'things', 'those', 'to', 'wait.'] >>> sorted_words =
ex25.sort_sentence(sentence)

>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_and_last(sentence) All wait.
>>> ex25.print_first_and_last_sorted(sentence) All who

```

As you go through each of these lines, make sure you can find the function that's being run in `ex25.py` and understand how each one works. If you get different results or error, you'll have to go fix your code, exit `python3.6`, and start over.

Study Drills

Take the remaining lines of the *What You Should See* output and figure out what they are doing.

Make sure you understand how you are running your functions in the `ex25` module.

Try doing this: `help(ex25)` and also `help(ex25.break_words)`. Notice how you get help for your module and how the help is those odd `"""` strings you put after each function in `ex25`? Those special strings are called *documentation comments*, and we'll be seeing more of them.

Typing `ex25.` is annoying. A shortcut is to do your import like this: `from ex25 import *`. This is like saying, "Import everything from `ex25`."

Programmers like saying things backward.

Start a new session and see how all your functions are right there.

Try breaking your file and see what it looks like in `python` when you use it. You will have to quit `python` with `quit()` to be able to reload it.

EVEN MORE PRACTICE 93

Common Student Questions

I get None printed out for some of the functions. You probably have a function that is missing the return at the end. Go backward through the file and confirm that every line is right.

I get -bash: import: command not found when I type import ex25. Pay attention to what I'm doing in the *What You Should See* section. I'm doing this in *Python*, not in the Terminal. That means you first run Python.

I get ImportError: No module named ex25.py when I type import ex25.py. Don't add the .py to the end. Python knows the file ends in .py, so you just type import ex25.

I get SyntaxError: invalid syntax when I run this. That means you have something like a missing (or " or similar syntax error on that line or above it. Any time you get that error, start at the line it mentions and check that it's right, then go backward checking each line above that.

How can the words.pop function change the words variable? That's a complicated question, but in this case words is a list, and because of that you can give it commands, and it'll retain the results of those commands. This is similar to how files and many other things worked when you were working with them.

When should I print instead of return in a function? The return from a function gives the line of code that called the function a result. You can think of a function as taking input through its arguments and returning output through return. The print is *completely* unrelated to this and only deals with printing output to the Terminal.

EXERCISE 26

This page intentionally left blank

EXERCISE 27 Memorizing Logic

T

oday is the day you start learning about logic. Up to this point you have done

everything you possibly can reading and writing files to the Terminal, and have learned quite a lot of the math capabilities

of Python.

From now on, you will be learning *logic*. You won't learn complex theories that academics love to study but just the basic logic that makes real programs work and that real programmers need every day.

Learning logic has to come after you do some memorization. I want you to do this exercise for an entire week. Do not falter. Even if you are bored out of your mind, keep doing it. This exercise has a set of logic tables you must memorize to make it easier for you to do the later exercises.

I'm warning you this won't be fun at first. It will be downright boring and tedious, but this teaches you a very important skill you will need as a programmer. You *will* need to be able to memorize important concepts in your life. Most of these concepts will be exciting once you get them. You will struggle with them, like wrestling a squid, then one day you will understand it. All that work memorizing the basics pays off big later.

Here's a tip on how to memorize something without going insane: Do a tiny bit at a time throughout the day and mark down what you need to work on most. Do not try to sit down for two hours straight and memorize these tables. This won't work. Your brain will only retain whatever you studied in the first 15 or 30 minutes anyway. Instead, create a bunch of index cards with each column on the left (True or False) on the front, and the column on the right on the back. You should then take them out, see the "True or False" and immediately say "True!" Keep practicing until you can do this.

Once you can do that, start writing out your own truth tables each night into a notebook. Do not just copy them. Try to do them from memory. When you get stuck, glance quickly at the ones I have here to refresh your memory. Doing this will train your brain to remember the whole table.

Do not spend more than one week on this, because you will be applying it as you go.

The Truth Terms

In Python we have the following terms (characters and phrases) for determining if something is “True” or “False.” Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

and
or
not

MEMORIZING LOGIC

!= (not equal)
== (equal)
>= (greater-than-equal)
<= (less-than-equal)
True
False

You actually have run into these characters before but maybe not the terms. The terms (and, or, not) actually work the way you expect them to, just like in English.

The Truth Tables

We will now use these characters to make the truth tables you need to memorize.

NOT True? not True False

not False True

OR True?

True or True False

True or True True

False or True True

False or False False

AND True?

True and False False

True and True True

False and False True

FalseandFalse False

NOT OR True?

not (True or False) False

not (True or True) False

not (False or False) True

not (False or True) False

NOT AND True?

not (True and True) False

not (True and False) True

not (False and True) True

not (False and False) True

!= True?

1 != 0 True

1 != 1 False

0 != 1 True

0 != 0 False

== True?

1 == 0 False 1 == 1 True

0 == 1 False

0 == 0 True

Now use these tables to write up your own cards and spend the week memorizing them. Remember, though, there is no failing in this book, just trying as hard as you can each day, and then a *little* bit more.

EXERCISE 28

Boolean Practice

T

he logic combinations you learned from the last exercise are called *Boolean logic* expressions.

Boolean logic is used *everywhere* in programming. It is a fundamental part of computation, and knowing the expressions is akin to knowing your scales in music.

In this exercise you will take the logic exercises you memorized and start trying them out in Python. Take each of these logic problems and write what you think the answer will be. In each case it will be either True or False.

Once you have the answers written down, start Python in your Terminal and type each logic problem in to confirm your answers.

True and True

False and True

1 == 1 and 2 == 1

"test" == "test"

1 == 1 or 2 != 1

True and 1 == 1

False and 0 != 0

True or 1 == 1

"test" == "testing"

1 != 0 and 2 == 1

"test" != "testing"

"test" == 1

not (True and False)

not (1 == 1 and 0 != 1)

not (10 == 1 or 1000 == 1000)

not (1 != 10 or 3 == 4)

not ("testing" == "testing" and "Zed" == "Cool Guy")

1 == 1 and (not ("testing" == 1 or 1 == 0))

"chunky" == "bacon" and (not (3 == 4 or 3 == 3))

3 == 3 and (not ("testing" == "testing" or "Python" == "Fun"))

BOOLEAN PRACTICE

I will also give you a trick to help you figure out the more complicated ones toward the end.

Whenever you see these Boolean logic statements, you can solve them easily by this simple process:

Find an equality test (== or !=) and replace it with its truth. Find each and/or inside parentheses and solve those first. Find each not and invert it.

Find any remaining and/or and solve it.

When you are done you should have True or False.

I will demonstrate with a variation on item 20:

3 != 4 and not ("testing" != "test" or "Python" == "Python")

Here's me going through each of the steps and showing you the translation

until I've boiled it down to a single result:

Solve each equality test:

3 != 4 is True: True and not ("testing" != "test" or "Python" == "Python")

"testing" != "test" is True: True and not (True or "Python" == "Python")

"Python" == "Python": True and not (True or True)

Find each and/or in parentheses ():

(True or True) is True: True and not (True) Find each not and invert it:

not (True) is False: True and False

Find any remaining and/ors and solve them: True and False is False

With that we're done and know the result is False.

primed for more of these "logic gymnastics" so that later cool stuff is much easier. Just stick with it, and keep track of what you get wrong, but do not worry that it's not getting in your head quite yet. It'll come.

What You Should See

After you have tried to guess at these, this is what your session with Python might look like:

```
$ python3.6
```

```
Python 2.5.1 (r251:54863, Feb 6 2009, 19:02:12)
```

```
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information. >>>
```

```
True and True
```

```
True
```

```
>>> 1 == 1 and 2 == 2
```

```
True
```

EXERCISE 29 What If

H



Here is the next script of Python you will enter, which introduces you to the if-statement. Type this in, make it run exactly right, and then we'll see if your practice has paid off. ex29.py

```
people = 20
```

```
cats = 30
```

```

dogs = 15
4
5
if people < cats:
print("Too many cats! The world is doomed!") 8
if people > cats:
print("Not many cats! The world is saved!") 11
if people < dogs:
print("The world is drooled on!")
14
if people > dogs:
print("The world is dry!")
17
18
19 dogs += 5
20
if people >= dogs:
print("People are greater than or equal to dogs.") 23
if people <= dogs:
print("People are less than or equal to dogs.") 26
27
if people == dogs:
print("People are dogs.")

```

What You Should See
Exercise 29 Session

```
$ python3.6 ex29.py
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs. People are dogs. WHAT IF
```

EXERCISE 30 Else and If

I

n the last exercise you worked out some if-statements and then tried to guess what they are and how they work. Before you learn more I'll explain what

everything is by answering the questions you

had from the previous *Study Drills* section. You did the *Study Drills*, right?

What do you think the if does to the code under it? An if-statement creates what is called a “branch” in the code. It’s kind of like those choose your own adventure books where you are asked to turn to one page if you make one choice and another if you go a different direction. The if-statement tells your script, “If this Boolean expression is True, then run the code under it; otherwise, skip it.”

Why does the code under the if need to be indented four spaces? A colon at the end of a line is how you tell Python you are going to create a new “block” of code, and then indenting four spaces tells Python what lines of code are in that block. This is *exactly* the same thing you did when you made functions in the first half of the book.

What happens if it isn’t indented? If it isn’t indented, you will most likely create a Python error.

Python expects you to indent *something* after you end a line with a : (colon).

Can you put other Boolean expressions from Exercise 27 in the if statement? Try it. Yes, you can, and they can be as complex as you like, although really complex things generally are bad style.

What happens if you change the initial values for people, cats, and dogs? Because you are comparing numbers, if you change the numbers, different if-statements will evaluate to True and the blocks of code under them will run. Go back and put different numbers in and see if you can figure out in your head which blocks of code will run.

Compare my answers to your answers, and make sure you *really* understand the concept of a “block” of code. This is important for when you do the next exercise where you write all the parts of if-statements that you can use.

Type this one in and make it work too. ex30.py

people = 30

```

cars = 40
trucks = 15
4
5
if cars > people:
print("We should take the cars.")
ELSE AND IF
elif cars < people:
print("We should not take the cars.") 10 else:
11 print("We can't decide.")
12
if trucks > cars:
print("That's too many trucks.") 15 elif trucks < cars: 16 print("Maybe we
could take the trucks.") 17 else: 18 print("We still can't decide.")
19
if people > trucks:
print("Alright, let's just take the trucks.") 22 else: 23 print("Fine, let's stay
home then.") What You Should See
Exercise 30 Session

```

\$ python3.6 ex30.py

We should take the cars.

Maybe we could take the trucks. Alright, let's just take the trucks.

EXERCISE 31 Making Decisions

I

In the first half of this book you mostly just printed out strings and called functions, but everything was basically in a straight line. Your scripts ran starting at the top and went to the bottom where they ended. If you made a function, you could run that function later, but it still didn't have the kind of branching you need to really make decisions. Now that you have if, else, and elif you can start to make scripts that decide things. In the last script you wrote out a simple set of tests asking some questions. In this script you will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out.

ex31.py

```

1 print("""You enter a dark room with two doors. 2 Do you go through door
#1 or door #2?""")

3
4 door = input("> ")
5
6 if door == "1":
7     print("There's a giant bear here eating a cheese cake.")
8     print("What do you do?")
9     print("1. Take the cake.")
10    print("2. Scream at the bear.")
11
12    bear = input("> ")
13
14    if bear == "1":
15        print("The bear eats your face off. Good job!")
16    elif bear == "2":
17        print("The
bear eats your legs off. Good job!")
18    else:
19        print(f"Well, doing {bear} is probably better.")
20        print("Bear runs away.")
21
22    elif door == "2":
23        print("You stare into the endless abyss at Cthulhu's retina.")
24        print("1.
Blueberries.")
25        print("2. Yellow jacket clothespins.")
26        print("3. Understanding revolvers yelling melodies.")
27
28    insanity = input("> ")
29
30    if insanity == "1" or insanity == "2":
31        print("Your body survives powered by a mind of jello.")
32        print("MAKING DECISIONS")
33    else:
34        print("The insanity rots your eyes into a pool of muck.")
35        print("Good job!")
36
37    else:

```

```
print("You stumble around and fall on a knife and die. Good job!")
```

A key point here is that you are now putting the if-statements *inside* ifstatements as code that can run. This is very powerful and can be used to create “nested” decisions, where one branch leads to another and another. Make sure you understand this concept of if-statements inside ifstatements. In fact, do the

Study Drills to really nail it.

What You Should See

Here is me playing this little adventure game. I do not do so well. Exercise 31 Session

```
$ python3.6 ex31.py
```

```
You enter a dark room with two doors. Do you go through door #1 or door #2?
```

```
> 1
```

```
There's a giant bear here eating a cheese cake.
```

```
What do you do?
```

```
Take the cake.0
```

```
Scream at the bear.
```

```
> 2
```

```
The bear eats your legs off. Good job!
```

EXERCISE 32 Loops and Lists

Y

ou should now be able to do some programs that are much more interesting. If you have been keeping up, you should realize that now you can combine all the other things you have learned with

if-statements and Boolean expressions to make your programs do smart things.

However, programs also need to do repetitive things very quickly. We are going to use a for-loop in this exercise to build and print various lists. When you do the exercise, you will start to figure out what they are. I won't tell you right now. You have to figure it out.

Before you can use a for-loop, you need a way to *store* the results of loops somewhere. The best way to do this is with lists. A list is exactly what the name says: a container of things that are organized in order from first to last. It's not complicated; you just have to learn a new syntax. First, here's how you make lists:

```
hairs = ['brown', 'blond', 'red'] eyes = ['brown', 'blue', 'green'] weights = [1, 2, 3, 4]
```

You start the list with the [(left bracket) which “opens” the list. Then you put each item you want in the list separated by commas, similar to function arguments. Lastly, end the list with a] (right bracket) to indicate that it's over. Python then takes this list and all its contents and assigns them to the variable.

WARNING! This is where things get tricky for people who can't code. Your brain has been taught that the world is flat. Remember in the last exercise where you put if-statements inside if-statements? That probably made your brain hurt because most people do not ponder how to “nest” things inside things. In programming nested structures are all over the place. You will find functions that call other functions that have if-statements that have lists with lists inside lists. If you see a structure like this that you can't figure out, take out a pencil and paper and break it down manually bit by bit until you understand it.

We now will build some lists using some for-loops and print them out:

```
ex32.py
```

```
the_count = [1, 2, 3, 4, 5]
```

```
fruits = ['apples', 'oranges', 'pears', 'apricots'] 3 change = [1, 'pennies', 2, 'dimes', 3, 'quarters']
```

```
4
```

```
# this first kind of for-loop goes through a list
```

LOOPS AND LISTS

```
for number in the_count:
```

```
print(f"This is count {number}")
```

```
8
```

```
9 # same as above 10 for fruit in fruits:
```

```

11 print(f"A fruit of type: {fruit}")
12
# also we can go through mixed lists too

# notice we have to use {} since we don't know what's in it 15for i in change:
16 print(f"I got {i}")

17
# we can also build lists, first start with an empty one
elements = []
20

21 # then use the range function to do 0 to 5 counts 22 for i in range(0, 6):

print (f"Adding {i} to the list.")
# append is a function that lists understand
elements.append(i)
26

27 # now we can print them out too 28 for i in elements: 29 print(f"Element
was: {i}")

```

What You Should See
Exercise 32 Session

```

$ python3.6 ex32.py This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples A fruit of type: oranges A fruit of type: pears A fruit of
type: apricots I got 1
I got pennies
I got 2
I got dimes
I got 3
I got quarters
Adding 0 to the list. Adding 1 to the list. Adding 2 to the list. Adding 3 to the
list.

```

Adding 4 to the list. Adding 5 to the list. Element was: 0 Element was: 1
Element was: 2 Element was: 3 Element was: 4 Element was: 5

EXERCISE 33 While Loops

N

ow to totally blow your mind with a new loop, the while-loop. A whileloop will keep executing the code block under it as long as a Boolean expression is True.

Wait, you have been keeping up with the terminology, right? That if we write a line and end it with a : (colon) then that tells Python to start a new block of code? Then we indent and that's the new code. This is all about structuring your programs so that Python knows what you mean. If you do not get that idea then go back and do some more work with ifstatements, functions, and the for-loop until you get it.

Later on we'll have some exercises that will train your brain to read these structures, similar to how we burned Boolean expressions into your brain.

Back to while-loops. What they do is simply do a test like an ifstatement, but instead of running the code block *once*, they jump back to the "top" where the while is and repeat. A while-loop runs until the expression is False.

Here's the problem with while-loops: Sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise, you almost always want your loops to end eventually.

To avoid these problems, there are some rules to follow:

Make sure that you use while-loops sparingly. Usually a for-loop is better.

Review your while statements and make sure that the Boolean test will become False at some point.

When in doubt, print out your test variable at the top and bottom of the while-loop to see what it's doing.

In this exercise, you learn the while-loop while doing these three checks:
ex33.py

```

i = 0
numbers = []
3
while i < 6:
print(f"At the top i is {i}")
numbers.append(i)
7
i = i + 1
print("Numbers now: ", numbers)

```

WHILE LOOPS

```

print(f"At the bottom i is {i}")
11
12
13 print("The numbers: ")
14

```

15 for num in numbers: 16 print(num) What You Should See

Exercise 33 Session

\$ python3.6 ex33.py

At the top i is 0

Numbers now: [0]

At the bottom i is 1

At the top i is 1

Numbers now: [0, 1]

At the bottom i is 2

At the top i is 2

Numbers now: [0, 1, 2]

At the bottom i is 3

At the top i is 3

Numbers now: [0, 1, 2, 3]

At the bottom i is 4

At the top i is 4

Numbers now: [0, 1, 2, 3, 4] At the bottom i is 5

At the top i is 5

Numbers now: [0, 1, 2, 3, 4, 5] At the bottom i is 6 The numbers: 0

1

2

3 4 5 EXERCISE 34 Accessing Elements of Lists

L

lists are pretty useful, but unless you can get at the things in them, they aren't all that good. You can already go through the elements of a list in order, but what if you want say, the fifth element? You need to know how to access the elements of a list. Here's how you would access the *first* element of a list:

```
animals = ['bear', 'tiger', 'penguin', 'zebra'] bear = animals[0]
```

You take a list of animals, and then you get the first (1st) one using 0?! How does that work? Because of the way math works, Python starts its lists at 0 rather than 1. It seems weird, but there are many advantages to this, even though it is mostly arbitrary.

The best way to explain why is by showing you the difference between how you use numbers and how programmers use numbers.

Imagine you are watching the four animals in our list (['bear', 'tiger', 'penguin', 'zebra']) run in a race. They cross the finish line in the *order* we have them in this list. The race was really exciting because the animals didn't eat each other and somehow managed to run a race. Your friend shows up late and wants to know who won. Does your friend say, "Hey, who came in *zeroth*?" No, he says, "Hey, who came in *first*?"

This is because the *order* of the animals is important. You can't have the second animal without the first (1st) animal, and you can't have the third without the second. It's also impossible to have a "zeroth" animal, since zero means nothing. How can you have a nothing win a race? It just doesn't make sense. We call these kinds of numbers "ordinal" numbers, because they indicate an ordering of things.

Programmers, however, can't think this way, because they can pick any element out of a list at any point. To programmers, the list of animals is more like a deck of cards. If they want the tiger, they grab it. If they want the zebra, they can take it, too. This need to pull elements out of lists at random means that they need a way to indicate elements consistently by an address, or an "index," and the best way to do that is to start the indices at 0. Trust me on

this: the math is way easier for these kinds of accesses. This kind of number is a “cardinal” number and means you can pick at random, so there needs to be a 0 element.

How does this help you work with lists? Simple: every time you say to yourself, “I want the third animal,” you translate this “ordinal” number to a “cardinal” number by subtracting 1. The “third” animal is at index 2 and is the penguin. You have to do this because you have spent your whole life using ordinal numbers, and now you have to think in cardinal. Just subtract 1 and you’re good.

Remember: ordinal == ordered, 1st; cardinal == cards at random, 0.

ACCESSING ELEMENTS OF LISTS

Let’s practice this. Take this list of animals, and follow the exercises where I tell you to write down what animal you get for that ordinal or cardinal number. Remember, if I say “first,” “second,” then I’m using ordinal, so subtract 1. If I give you cardinal (like “The animal at 1”), then use it directly.

animals = ['bear', 'python3.6', 'peacock', 'kangaroo', 'whale', 'platypus'] The animal at 1.

The third (3rd) animal.

The first (1st) animal.

The animal at 3.

The fifth (5th) animal.

The animal at 2.

The sixth (6th) animal.

The animal at 4.

For each of these, write out a full sentence of the form: “The first (1st) animal is at 0 and is a bear.” Then say it backwards: “The animal at 0 is the 1st animal and is a bear.” Use your Python to check your answers.

Study Drills

With what you know of the difference between these types of numbers, can you explain why the year 2010 in “January 1, 2010,” really is 2010 and not 2009? (Hint: you can’t pick years at random.)

Write some more lists and work out similar indexes until you can translate them.

Use Python to check your answers.

WARNING! Programmers will tell you to read this guy named *Dijkstra* on this subject. I recommend you avoid his writings on this unless you enjoy being yelled at by someone who stopped programming at the same time programming started.

EXERCISE 35 Branches and Functions

You have learned if-statements, functions, and lists. Now it's time to bend your mind. Type this in, and see if you can figure out what it's doing.

ex35.py

```
1 from sys import exit
2
3 def gold_room():
4     print("This room is full of gold. How much do you take?")
5
6     choice = input("> ")
7     if "0" in choice or "1" in choice:
8         how_much = int(choice)
9     else:
10        dead("Man, learn to type a number.")
11
12    if how_much < 50:
13        print("Nice, you're not greedy, you win!")
14        exit(0)
15    else:
16        dead("You greedy bastard!")
17
18    def bear_room():
19        print("There is a bear here.")
20        print("The bear has a bunch of honey.")
21        print("The fat bear is in front of another door.")
22        print("How are you going to move the bear?")
23        bear_moved = False
```

```
25
while True:
choice = input("> ")
28
if choice == "take honey":

dead("The bear looks at you then slaps your face off.") 31 elif choice ==
"taunt bear" and not bear_moved:

print ("The bear has moved from the door.")
print("You can go through it now.")
bear_moved = True
elif choice == "taunt bear" and bear_moved:

dead("The bear gets pissed off and chews your leg off.") 37 elif choice ==
"open door" and bear_moved:
```

```
38 gold_room() 39 else:
40 print("I got no idea what that means.")
41
```

BRANCHES AND FUNCTIONS

```
42
def cthulhu_room():

print ("Here you see the great evil Cthulhu.")
print("He, it, whatever stares at you and you go insane.") print("Do you flee
for your life or eat your head?") 47
48 choice = input("> ")
49
if "flee" in choice:
start()
elif "head" in choice: 53 dead("Well that was tasty!") else:
cthulhu_room()
56
57
def dead(why):
print(why, "Good job!")
exit(0)
```



```

61
def start():
    print("You are in a dark room.")
    print("There is a door to your right and left.")
    print("Which one do you take?")
66
67 choice = input("> ")
if choice == "left":
    bear_room()
elif choice == "right":
    cthulhu_room() 73 else:
74 dead("You stumble around the room until you starve.") 75
76
77 start()

```

What You Should See

Here's me playing the game:

Exercise 35 Session

```
$ python3.6 ex35.py
```

You are in a dark room.

There is a door to your right and left. Which one do you take?

```
> left
```

There is a bear here.

The bear has a bunch of honey. The fat bear is in front of another door.

How are you going to move the bear? > taunt bear

The bear has moved from the door. You can go through it now.

```
> open door
```

This room is full of gold. How much do you take?

```
> 1000
```

You greedy bastard! Good job!

Study Drills

Draw a map of the game and how you flow through it.

Fix all of your mistakes, including spelling mistakes.

Write comments for the functions you do not understand.

Add more to the game. What can you do to both simplify and expand it?

The gold_room has a weird way of getting you to type a number. What are all the bugs in this way of doing it? Can you make it better than what I've written? Look at how int() works for clues.

Common Student Questions

Help! How does this program work?! When you get stuck understanding a piece of code, simply write an English comment above *every* line explaining what that line does. Keep your comments short and similar to the code. Then either diagram how the code works or write a paragraph describing it. If you do that you'll get it.

Why did you write while True? That makes an infinite loop.

What does exit(0) do? On many operating systems a program can abort with exit(0), and the number passed in will indicate an error or not. If you do exit(1) then it will be an error, but exit(0) will be a good exit. The reason it's backward from normal Boolean logic (with 0==False) is that you can use different numbers to indicate different error results. You can do exit(100) for a different error result than exit(2) or exit(1).

Why is input() sometimes written as input('> ')? The parameter to input is a string that it should print as a prompt before getting the user's input.

XERCISE 36 Designing and Debugging

N

Now that you know if-statements, I'm going to give you some rules for for-loops and while-loops that will keep you out of trouble. I'm also going to give you some tips on debugging so that you can figure out problems with your program. Finally, you will design a little game similar to the last exercise but with a slight twist.

Rules for if-statements

Every if-statement must have an else.

If this else should never run because it doesn't make sense, then you must use a die function in the else that prints out an error message and dies, just like

we did in the last exercise. This will find *many* errors.

Never nest if-statements more than two deep and always try to do them one deep.

Treat if-statements like paragraphs, where each if-elif-else grouping is like a set of sentences. Put blank lines before and after.

Your Boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

If you follow these simple rules, you will start writing better code than most programmers. Go back to the last exercise and see if I followed all of these rules. If not, fix my mistakes.

WARNING! Never be a slave to the rules in real life. For training purposes you need to follow these rules to make your mind strong, but in real life sometimes these rules are just stupid. If you think a rule is stupid, try not using it. Rules for Loops

Use a while-loop only to loop forever, and that means probably never. This only applies to Python; other languages are different.

Use a for-loop for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

DESIGNING AND DEBUGGING

Tips for Debugging

Do not use a “debugger.” A de bugger is like doing a full-body scan on a sick person. You do not get any specific useful information, and you find a whole lot of information that doesn’t help and is just confusing.

The best way to debug a program is to use print to print out the values of variables at points in the program to see where they go wrong.

Make sure parts of your programs work as you work on them. Do not write massive files of code before you try to run them. Code a little, run a little, fix a little.

Homework

Now write a game similar to the one that I created in the last exercise. It can be any kind of game you want in the same flavor. Spend a week on it making it as interesting as possible. For Study Drills, use lists, functions, and modules (remember those from Exercise 13?) as much as possible, and find as many new pieces of Python as you can to make the game work.

Before you start coding you must draw a map for your game. Create the rooms, monsters, and traps that the player must go through on paper before you code.

Once you have your map, try to code it up. If you find problems with the map then adjust it and make the code match.

The best way to work on a piece of software is in small chunks like this: On a sheet of paper or an index card, write a list of tasks you need to complete to finish the software. This is your to-do list.

Pick the easiest thing you can do from your list.

Write out English comments in your source file as a guide for how you would accomplish this task in your code.

Write some of the code under the English comments.

Quickly run your script so see if that code worked.

Keep working in a cycle of writing some code, running it to test it, and fixing it until it works.

Cross this task off your list, then pick your next easiest task and repeat.

This process will help you work on software in a methodical and consistent manner. As you work, update your list by removing tasks you don't really need and adding ones you do.

3 STORING AND MODIFYING INFORMATION

introduces you to CRUD, Create, Read, Update, and Delete

.

not that Chapter 3 contains cruddy material. This acronym provides an easy method to remember precisely what tasks all computer programs perform with information you want to manage. Of

course, geeks use a special term for information
.data, but either information or data works fine for this book.

To make information useful, you have to have some means of storing it permanently. Otherwise, every time you turned the computer off, all your information would be gone and the computer would provide limited value. In addition, Python must provide some rules for modifying information. The alternative is to have applications running amok, changing information in any and every conceivable manner. This chapter is about

controlling information
.defining how information is stored permanently

and manipulated by applications you create. You can find the source code for this chapter in the BPPD_06_Storing_And_Modifying_Information.ipynb file provided with the downloadable source code, as described

.Storing Information

An application requires fast access to information or else it will take a long time to complete tasks. As a result, applications store information in memory. However, memory is temporary. When you turn off the machine, the information must be stored in some permanent form, such as on your hard drive, a Universal Serial Bus (USB) flash drive, a Secure Digital (SD) card, or on the Internet using something like Anaconda Cloud. In addition, you must also consid

number or text. The following sections discuss the issue of storing information as part of an application in more detail.

Seeing variables as storage boxes

When working with applications, you store information in variables. A *variable* is a kind of storage box. Whenever you want to work with the information, you access it using the variable. If you have new information

you want to store, you put it in a variable. Changing information means accessing the variable first and then storing the new value in the variable.

Just as you store things in boxes in the real world, so you store things in variables (a kind of storage box) when working with applications.

Computers are actually pretty tidy. Each variable stores just one piece of information. Using this technique makes it easy to find the particular piece

of information you need

unlike in your closet, where things from ancient

Egypt could be hidden. Even though the examples you work with in previous chapt

variables to make working with information easier.

Using the right box to store the data

People tend to store things in the wrong sort of box. For example, you might find a pair of shoes in a garment bag and a supply of pens in a shoebox. However, Python likes to be neat. As a result, you find numbers stored in one sort of variable and text stored in an entirely different kind of variable. Yes, you use variables in both cases, but the variable is designed to store a particular kind of information. Using specialized variables makes it possible to work with the information inside in

particular ways. You

just keep in mind that each kind of information is stored in a special kind of variable.

Python uses specialized variables to store information to make things easy for the programmer and to ensure that the information remains safe.

that the computer knows about are 0s and 1s, which is the absence or presence of a voltage. At a higher level, computers do work with numbers,

times, and any other kind of information you can think about all come down to 0s and 1s in the computer system. For example, the letter A is actually stored as 01000001 or the number 65. The computer has no concept of the letter A or of a date such as 8/31/2017.

Defining the Essential Python Data Types

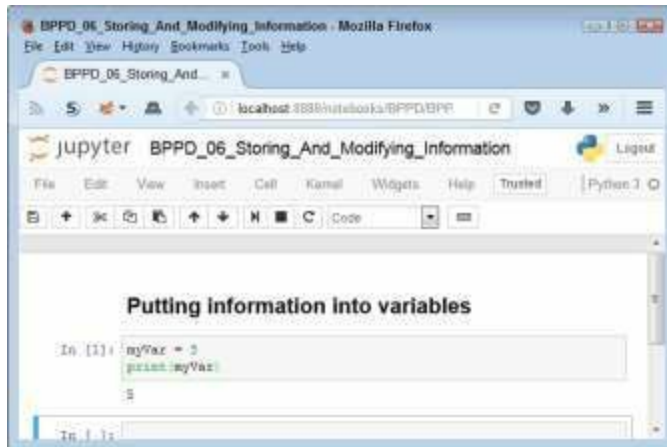
Every programming language defines variables that hold specific kinds of information, and Python is no exception. The specific kind of variable is called a *datatype*. Knowing the data type of a variable is important because it tells you what kind of information you find inside. In addition, when you want to store information in a variable, you need a variable of

variable designed to hold numeric information. Doing so would damage the text and cause problems with the application. You can generally classify Python data types as numeric, string, and Boolean, although there

sections describe each of the standard Python data types within these classifications.

Putting information into variables

To place a value into any variable, you make an assignment using the assignment operator (=). Chapter 7 discusses the whole range of basic



Python operators in more detail, but you need to know how to use this particular operator to some extent now. For example, to place the number 5 into a variable named myVar, you type myVar = 5 and press Enter at the Python prompt. Even

, you can always type the variable name and press Enter to see the value it contains, as shown in Figure 6-1.

FIGURE 6-1:

Use the assignment operator to place information into a variable.

Understanding the numeric types

Humans tend to think about numbers in general terms. We view 1 and 1.0 as being the same number one of them simply has a decimal point.

could easily use them interchangeably. Python views them as being different kinds of numbers because each form requires a different kind of processing. The following sections describe the integer, floating-point, and complex number classes of data types that Python supports.

Integers

Any whole number is an *integer*. For example, the value 1 is a whole number by the int data type.

As with storage boxes, variables have capacity limits. Trying to stuff a value

you can store numbers between
9,223,372,036,854,775,808 and

9,223,372,036,854,775,807 within an int (which is the maximum value that fits in a 64-bit variable). Even tho

.

infinite.

When working with the int type, you have access to a number of interesting features. Many of them appear later in the book, but one feature is the ability to use different numeric bases:

Uses only 0

and 1 as numbers.

Uses the numbers 0 through 7.

Uses the

Base 8: usual numeric system.

Is also called

the letters A

values.

hex and uses the numbers 0 through 9 and through F to create 16 different possible

To tell

you add a 0

.

example,

Python when to use bases other than base 10, and a special letter to the number. For 0b100 is the value one-zero-zero in base 2.

Here are the letters you normally use:

.Base 2
.Base 8
.Base 16

You can also convert numeric values to other bases by using the `bin()`, `oct()`, and `hex()` commands. So, putting everything together, you can see how to convert between bases using the commands shown in Figure 6-2. Try the command shown in the figure yourself so that you can see how the various bases work. Using a different base actually makes things easier in many

book. For now, all you really need to know is that integers support different numeric bases.

Understanding the numeric types

Integers

```
In [2]: Test = 0b100
print("100 Binary: ", Test)

Test = 0o100
print("100 Octal: ", Test)

Test = 100
print("100 Decimal: ", Test)

Test = 0x100
print("100 Hexadecimal: ", Test)

print(bin(Test))
print(oct(Test))
print(hex(Test))

100 Binary: 4
100 Octal: 64
100 Decimal: 100
100 Hexadecimal: 204
0b100000000
0o400
0x100
```

FIGURE 6-2:

Integers have many interesting features, including the capability to use different numeric

bases.

Floating-point values

Any number that includes a decimal portion is a floating-point value. For example, 1.0 has a decimal

par

-point value. Many people get confused about whole

numbers and floating-point numbers, but the difference is easy to

.

-point

value. Python stores floating-point values in the float data type.

Floating-point values have an advantage over integer values in that you can store immensely large or incredibly small values in them. As with integer variables, floating-point variables have a storage capacity. In their case, the maximum value that a variable can contain is

1.7976931348623157

10^{308} and the minimum value that a variable can contain is

2.2250738585072014

10^{-308} on most platforms. When working with floating-point values, you can assign the information to the variable in a number of ways. The two most common methods are to provide the number directly and to use scientific notation. When using scientific notation, an *e* separates the number from its exponent. Figure 6-3 shows both methods of making an assignment. Notice that using a negative exponent results in a fractional value.

UNDERSTANDING THE NEED FOR MULTIPLE NUMBER TYPES

A lot of new developers (and even some older ones) have a hard time understanding why there is a need for more than one numeric type. After all, humans can use just one kind of number. To understand the need for multiple number types, you have to understand a little about how a computer works with numbers.

An integer is stored in the computer as simply a series of bits that the computer reads directly. A value of 0100 in binary equates to a value of 4 in decimal. On the other hand, numbers that have decimal points are stored in an entirely different manner. Think back to all those classes

you slept through on exponents in school
they actually

come in handy sometimes. A floating-point number is stored as a sign bit (plus or minus), *mantissa* (the fractional part of the number), and

exponent(the power of 2). (Some

texts use the term *significand* in place of mantissa .the terms are interchangeable.) To obtain the floating-point value, you use the equation:

$$\text{Value} = \text{Mantissa} * 2^{\text{Exponent}}$$

At one time, computers all used different floating-point representations, but they all use the IEEE-754 standard now. You can read about this standard at <http://grouper.ieee.org/groups/754/>. A full explanation of precisely how floating-point numbers

work is outside the scope of this book, but you can read a fairly understandable description at

http://www.cprogramming.com/tutorial/floating_point/understanding_floating_point_representation.html. Nothing helps you

understand a concept like playing with the values. You can find a really interesting floating-point number converter at [http://www.h-schmidt.net/FloatConverter/ IEEE754.html](http://www.h-schmidt.net/FloatConverter/IEEE754.html), where you can click the individual bits (to turn them off or on) and see the floating-point number that results.

As you might imagine, floating-point numbers tend to consume more space in memory because of their complexity. In addition, they use an entirely different area

of the processor

.one that works more slowly than the

part used for integer math. Finally, integers are precise, as contrasted to floating

.

precisely represent some numbers, so you get an approximation instead. However, floating-point variables can store much larger numbers. The bottom line is that decimals are unavoidable in the real world, so you need floating-point numbers, but using integers when you can reduces the amount of memory your application consumes and helps it work

faster. Computer systems have many trade-offs, and this one is unavoidable.

```
Floating-point values
In [3]: Test = 255.0
print("Direct Assignment: ", Test)

Test = 2.55e2
print("Scientific Notation: ", Test)

Test = 2.55e-2
print("Negative Exponent: ", Test)

Direct Assignment: 255.0
Scientific Notation: 255.0
Negative Exponent: 0.0255
```

FIGURE 6-3:

Floating-point values provide multiple assignment techniques.

Complex numbers

You may or may not remember complex numbers from school. A *complex number* consists of a real number and an imaginary number that are paired

numbers, you can read about them at

<http://www.mathsisfun.com/numbers/complex-numbers.html>. Realworld uses for complex numbers include:

Electrical engineering

Complex

should give involved in

Python is one data type to book, you especially

The imaginary with a *jafter* number with part, you

myComplex =

If you want to type

.Fluid dynamics

.Quantum mechanics

.Computer graphics

.Dynamic systems

numbers have other uses, too, but this list .

any of these disciplines, you probably encounter complex numbers. However, of the few languages that provides a built-in support them. As you progress through the find other ways in which Python lends itself well to science and engineering.

part of a complex number always appears it. So, if you want to create a complex 3 as the real part and 4 as the imaginary make an assignment like this:

$3 + 4j$

see the real part of the variable, you simply `myComplex.real` at the Python prompt and press Enter. Likewise, if you want to see the imaginary part of the variable, you type `myComplex.imag` at the Python prompt and press Enter.

Sometimes you might want to know the variable type. Perhaps the type of a variable, use the `type()` method. For example, if you start by placing a value of 5 in `myInt` by typing `myInt = 5` and pressing Enter, you can find the type of `myInt` by typing `type(myInt)` and pressing Enter.

The output will be `<class 'int'>`, which means that `myInt` contains an int value.

Understanding Boolean values

It may seem amazing, but computers always give you a straight answer!

A

either True or False. In fact, there is an entire branch of mathematics called Boolean algebra that was originally defined by George Boole (a super-geek of his time) that computers rely upon to make decisions.

Contrary to

common belief, Boolean algebra has existed since 1854 long before the time of computers.

When using Boolean value in Python, you rely on the `bool` type. A variable of this type can contain only two values: True or False. You can

assign a value by using the `True` or `False` keywords, or you can create an expression that defines a logical idea that equates to true or false. For example, you could say, `myBool = 1 > 2`, which would equate to `False` because 1 is most definitely not greater than 2. You see the `bool` type used extensively in the

.Understanding strings

Of all the data types, strings are the most easily understood by humans and not understood at all by computers. If you have read the previous chapters in this book, you have already seen strings used quite often. For example, all the example code in Chapter 4 relies on strings. A *string* is simply any grouping of characters you place within double quotes. For example, `myString = "Python is a great language."` assigns a string of characters to `myString`.

by a number in memory. For example, the letter `A` is actually the number 65. To see this for yourself, type
at the Python prompt and press Enter. You see 65 as output. You can convert any single letter to its numeric equivalent by using the `ord()` command.

so useful in writing applications, you sometimes need to convert a string to a number. You can use the `int()` and `float()` commands to perform this

conversion. For example, if you type
and press Enter

at the Python prompt, you create an `int` named `myInt` that contains the value 123. Figure 6-4 shows how you can perform this task and validate the content and type of `myInt`.

FIGURE 6-4:

Converting number is `int()` and

Understanding strings

```
In [6]: print(ord("A"))
myInt = int("123")
print(myInt)
print(type(myInt))
65
123
<class 'int'>
```

a string to a easy by using the float()

commands.

You can convert numbers to a string as well by using the str() command. For example, if you type myStr = str(1234.56) and press Enter, you create a string containing the value "1234.56" and assign it to myStr. Figure 6-5 shows this type of conversion and the test you can perform on it. The point is that you can go back and forth between strings and numbers with great ease. Later chapters demonstrate how these conversions make a lot of seemingly impossible tasks quite doable.

FIGURE 6-5:

You can

```
In [7]: myStr = str(1234.56)
print(myStr)
print(type(myStr))
1234.56
<class 'str'>
```

convert numbers to strings as well.

Working with Dates and Times

Dates and times are items that most people work with quite a bit. Society bases almost everything on the date and time that a task needs to be or was completed. We make appointments and plan events for specific dates and times. Most of our day revolves around the clock. Because of the timeoriented nature of humans, i

.

with interacting with dates and time (especially storing these values for later use). As with everything else, computers understand only numbers

-

To work with dates and times, you need to perform a special task in Python. When writing computer books, chicken-and-egg scenarios always arise, and this is one of them. To use dates and times, you must

issue a special `import datetime` command. Technically, this act is called *importing*

a package,

the command works right now

just use it whenever you want to do something with date and time.

Computers do have clocks inside them, but the clocks are for the humans using the computer. Yes, some software also depends on the clock, but again, the emphasis is on human needs rather than anything the computer might require. To get the current time, you can simply type `datetime.datetime.now()` and press Enter. You see the full date and time

-6).

FIGURE 6-6:

Get the

Working with Dates and Times

```
In [8]: import datetime
        print(datetime.datetime.now())
2017-09-14 11:30:35.198385
```

current date and time by using the

`now()` command.

time by using the `now()` command. 7: Make the date

and time using the You may have noticed that the date and time are a little hard to read in the existing format. Say that you want to get just

a few things you discovered in previous sections to accomplish that task.

Type

`str(datetime.datetime.now().date())` and press Enter. Figure 6-7 shows that you now have something a little more usable.

```
In [9]: print(str(datetime.datetime.now().date()))
2017-09-14
```

more readable by `str()`

command.

Interestingly enough, Python also has a `time()` command, which you can use to obtain the current time. You can obtain separate values for each of the components that make up date and time by using the `day`, `month`,

year, hour, minute, second, and microsecond values. Later chapters help you understand how to use these various date and time features to keep application users informed about the current date and time on their system.

Chapter 7 Managing Information

W

hether you use the term *information* or *data* to refer to the content that applications manage, the fact is that you must provide some means of .Throughout the rest of the book, you see *information* and *data* used interchangeably because they really are the same thing, and in real-world

idea. No matter which term you use, you need some means of assigning data to variables, modifying the content of those variables to achieve specific goals, and comparing the result you receive with desired results. This chapter addresses all three requirements so that you can start to control data within your applications.

Also essential is to start working through methods of keeping your code understandable. Yes, you could write your application as a really long procedure, but trying to understand such a procedure is incredibly hard,

more than once. Functions are one way for you to package code so that you can more easily understand and reuse as needed.

Applications also need to interact with the user. Yes, some perfectly usable

order to provide a useful service, most applications interact with the user to discover how the user wants to manage data. You get an overview of this process in this chapter. Of course, you visit the topic of user interaction

Controlling How Python Views Data

As discussed in Chapter 6, all data on your computer is stored as 0s and 1s.

, Boolean values, dates, times, or any other kind of information except numbers. In addition,

-

depending on Python to translate the concept of a string into a form the computer can understand. The storage containers that your application creates and uses in the form of variables tell Python how to treat the 0s and 1s that the computer has stored. So, you need to understand that the

Python acts as an intermediary to make your applications functional.

To manage data within an application, the application must control the way in which Python views the data. The use of operators, packaging methods such as functions, and the introduction of user input all help applications control data. All these techniques rely, in part, on making comparisons. Determining what to do next means understanding what state the data is in now as compared to some other state. If the variable contains the name John now, but you really want it to contain Mary instead, then you first need to know that it does in fact contain John. Only then can you make the decision to change the content of the variable to Mary.

Making comparisons

operators. In fact, operators play a major role in manipulating data as well.

.

cusses how

operators work and how you can use them in applications to control data in

various ways. Later chapters use operators extensively as you discover techniques for creating applications that can make decisions, perform tasks

repetitively, and interact with the user in interesting ways. However, the basic idea behind operators is that they help applications perform various

types of comparisons.

In some cases, you use some fancy methods for performing comparisons in an application. For example, you can compare the output of two functions

chapter). With Python, you can perform comparisons at a number of levels so that you can manage data without a problem in your application. Using these techniques hides detail so that you can focus on the point of the comparison and define how to react to that comparison rather than become mired in detail. Your choice of techniques for performing comparisons affects the manner in which Python views the data and determines the sorts of things you can do to manage the data after the comparison is made. All this functionality might seem absurdly complex at the moment, but the important point to remember is that applications require comparisons in order to interact with data correctly.

Understanding how computers make comparisons

other structures that you create with Python. All this packaging is for your .

However, computers do directly support the concept of operators. Most Python operators have a direct corollary with a command that the computer understands directly. For example, when you ask whether one number is greater than another number, the computer can actually perform this computation directly, using an operator. (The upcoming section explains operators in detail.)

when you ask Python to compare two strings, what Python actually does is compare the numeric value of each character in the string. For example, the letter A is actually the number 65 in the computer. A lowercase letter a has

a different numeric value

.97. As a result, even though you might see ABC as being equal to abc .it sees them as different because the numeric values of their individual

letters are different.

Working with Operators

Operators are the basis for both control and management of data within applications. You use operators to define how one piece of data is compared to another and to modify the information within a single variable. In fact, operators are essential to performing any sort of math-related task and to assigning data to variables in the first place. When using an operator, you must supply either a variable or an expression. You already know that a variable is a kind of storage box used to hold data. An *expression* is an equation or formula that provides a description of a mathematical concept. In most cases, the result of evaluating an expression is a Boolean (true or false) value. The following sections describe operators in detail because you use them everywhere throughout the rest of the book.

Defining the operators

An *operator* accepts one or more inputs in the form of variables or expressions, performs a task (such as comparison or addition), and then provides an output consistent with that task. Operators are classified partially by their effect and partially by the number of elements they require. For example, a unary operator works with a single variable or expression; a binary operator requires two.

The elements provided as input to an operator are called *operands*. The operand on the left side of the operator is called the left operand, while the operand on the right side of the operator is called the right operand. The following list shows the categories of operators that you use within Python:

- .Unary
- .Arithmetic
- .Relational
- .Logical
- .Bitwise
- .Assignment
- .Membership
- .Identity

Each of these categories performs a specific task. For example, the arithmetic operators perform math-based tasks, while relational operators perform comparisons. The following sections describe the operators based on the category in which they appear.

UNDERSTAND

A ternary operator requires three elements. Python supports just one such operator, and you use it to determine the truth value of an expression. This ternary operator takes the following form (it apparently has no actual name, but you can call it the if. .else operator if desired):

TrueValue if Expression else FalseValue

When the Expression is true, the operator outputs TrueValue. When the expression is false, it outputs FalseValue. As an example, if you type

"Hello" if True else "Goodbye"

the operator outputs a response of 'Hello'. However, if you type

"Hello" if False else "Goodbye"

the operator outputs a response of 'Goodbye'. This is a handy operator for times when you need to make a quick .nt to write a lot of code to do it.

One of the advantages of using Python is that it normally has more than one way to do things. Python has an

alternative form of this ternary operator

an even shorter shortcut. It takes the following form:

(FalseValue, TrueValue)[Expression]

As before, when Expression is true, the operator outputs TrueValue; otherwise, it outputs FalseValue. Notice that the TrueValue and FalseValue elements are reversed in this case. An example of this version is

("Hello", "Goodbye")[True]

In this case, the output of the operator is 'Goodbye' because

forms, the first is a little clearer, while the second is shorter.

Unary

Unary operators require a single variable or expression as input. You often use these operators as part of a decision-making process. For example, you .

-1 shows the unary operators.

TABLE 7-1 Python Unary Operators

Operator	Description	Example

~	Inverts the bits in a number so that all the 0 bits become 1 bits and vice versa.	~4 results in a

	value of	
	5	

-	Negates the original value so that positive becomes negative and vice versa.	

.	(	
	4)	results in 4 and
	4	results in
	4	

+ Is provided purely for the sake of completeness. This operator returns the same value that you provide as input.

Arithmetic +4 results in a value of 4

Computers are known for their capability to perform complex math. However, the complex tasks that computers perform are often based on much simpler math tasks, such as addition. Python provides access to libraries that help you perform complex math tasks, but you can always create your own libraries of math functions using the simple operators found in Table 7-2.

TABLE 7-2 Python Arithmetic Operators

Operator Description Example

+ Adds two values together $5 + 2 = 7$

- Subtracts the right operand from the left operand $5 - 2 = 3$

*** Multiplies the right operand by the left operand $5 * 2 = 10$**

/ Divides the left operand by the right operand $5 / 2 = 2.5$

% Divides the left operand by the right operand and $5 \% 2 =$ returns the remainder 1

**** Calculates the exponential value of the right $5 ** 2 =$ operand by the left operand 25**

// Performs integer division, in which the left $5 // 2 =$ operand is divided by the right operand and only 2 the whole number is returned (also called floor division)

Relational

The relational operators compare one value to another and tell you when

1 is never greater than 2. The truth value of relations is often used to make decisions in your applications to ensure that the condition for performing a specific task is met.

Table 7-3 describes the relational operators.

TABLE 7-3 Python Relational Operators

Operator Description Example

== Determines whether two values are equal. Notice $1 == 2$ is that the relational operator uses two equals signs. False A mistake many developers make is using just one equals sign, which results in one value being assigned to another.

!= Determines whether two values are not equal. $1 != 2$ is Some older versions of Python allowed you to use True the $<>$ operator in place of the $!=$ operator. Using the $<>$ operator results in an error in current versions of Python.

> Verifies that the left operand value is greater than $1 > 2$ is the right

operand value. False

< Verifies that the left operand value is less than the 1 < 2 is right

operand value. True

>= Verifies that the left operand value is greater than 1 >= 2 is or equal to the right operand value. False

<= Verifies that the left operand value is less than or 1 <= 2 is equal to the right operand value. True Logical

The logical operators combine the true or false value of variables or expressions so that you can determine their resultant truth value. You use the logical operators to create Boolean expressions that help determine whether to perform tasks. Table 7-4 describes the logical operators.

TABLE 7-4 Python Logical Operators

Operator Description Example

and Determines whether both operands are true.

True and True is True

True and False is False

False and True is False

False and False is False

or Determines when one of two operands is true.

True or True is True

True or False is True

False or True is True

False or False is False

not Negates the truth value of a single operand. A true value becomes false and a false value becomes true.

not True is

False not False is True

Bitwise

The bitwise operators interact with the individual bits in a number. For example, the number 6 is actually 0b0110 in binary.

If your binary is a little rusty, you can use the handy Binary to Decimal to Hexadecimal Converter at <http://www.mathsisfun.com/binary-decimalhexadecimalconverter.html>. You need to enable JavaScript to make the site work.

A bitwise operator would interact with each bit within the number in a specific way. When working with a logical bitwise operator, a value of 0 counts as false and a value of 1 counts as true. Table 7-5 describes the bitwise operators.

TABLE 7-5 Python Bitwise Operators

Operator	Description	Example

&	(And) Determines whether both 0b1100 & 0b0110 individual bits within two = 0b0100 operators are true and sets the resulting bit to true when they are.	

	(Or) Determines whether either of 0b1100 0b0110	

	the individual bits within two = 0b1110 operators is true and sets the resulting bit to true when one of them is.	

^	(Exclusive or) Determines whether just one of 0b1100 ^ 0b0110	

	the individual bits within two = 0b1010 operators is true and sets the resulting bit to true when one is.	
	When both bits are true or both bits are false, the result is false.	

.		

	complement) ~0b1100 =	

-		

	0b1101	

complement value of a number. $\sim 0b0110 = .0b0111$

<< (Left shift) Shifts the bits in the left `0b00110011 << 2`

operand left by the value of the =

right operand. All new bits are `0b11001100` set to 0 and all bits that flow off the end are lost.

>> (Right shift) Shifts the bits in the left `0b00110011 >> 2`

operand right by the value of =

the right operand. All new bits `0b00001100` are set to 0 and all bits that flow off the end are lost.

Assignment

The assignment operators place data within a variable. The simple assignment operator appears in previous chapters of the book, but Python offers a number of other interesting assignment operators that you can use. These other assignment operators can perform mathematical tasks during the assignment process, which makes it possible to combine assignment with a math operation. Table 7-6 describes the assignment operators. For this particular table, the initial value of `MyVar` in the Example column is 5.

TABLE 7-6 Python Assignment Operators

Operator **Description** **Example**

= Assigns the value found in the right operand `MyVar = 5` to the left operand. results in `MyVar` containing 5

+= Adds the value found in the right operand to the value found in the left operand and places the result in the left operand. `MyVar += 2` results in `MyVar` containing 7

-= Subtracts the value found in the right operand from the value found in the left operand and places the result in the left operand.

**MyVar -= 2 results in
MyVar
containing 3**

***= Multiplies the value found in the right operand by the value found in the left operand and places the result in the left operand.
MyVar *= 2 results in
MyVar
containing 10**

**/= Divides the value found in the left operand by the value found in the right operand and places the result in the left operand. MyVar /= 2 results in
MyVar
containing 2.5**

**%= Divides the value found in the left operand by the value found in the right operand and places the remainder in the left operand. MyVar %= 2 results in
MyVar
containing 1**

****= Determines the exponential value found in the left operand when raised to the power of MyVar **= 2 results in
the value found in the right operand and places the result in the left operand. MyVar
containing 25**

**//= Divides the value found in the left operand by the value found in the right operand and places the integer (whole number) result in the left operand.
MyVar //= 2 results in
MyVar
containing 2**

Membership

The membership operators detect the appearance of a value within a list

or sequence and then output the truth value of that appearance. Think of the membership operators as you would a search routine for a database. You enter a value that you think should appear in the database, and the search routine finds it for you or reports

.

database. Table 7-7 describes the membership operators.

TABLE 7-7 Python Membership Operators

Operator Description Example

In Determines whether the value in the left

.

operand appears in the sequence found in the

.

right operand.

.

is True not in Determines whether the value in the left

.

operand is missing from the sequence found not in in the right operand.

is False

Identity

The identity operators determine whether a value or expression is of a

actually working with the sort of information that you think you are.

Using the identity operators can help you avoid errors in your application or determine the sort of processing a value requires. Table 7-8 describes the identity operators.

TABLE 7-8 Python Identity Operators

Operator Description Example

Is Evaluates to true when the type of the value or expression in the right operand points to the same type in the left operand. `type(2) is int` is True

is not Evaluates to true when the type of the value or expression in the right operand points to a different type than the value or expression in the left operand.

type(2) is not int is False

Understanding operator precedence

When you create simple statements that contain just one operator, the order of determining the output of that operator is also simple. However, when you start working with multiple operators, it becomes necessary to determine which operator to evaluate first. For example, you should know whether $1 + 2 * 3$ evaluates to 7 (where the multiplication is done first) or 9 (where the addition is done first). An order of operator precedence tells you that the answer is 7 unless you use parentheses to override the default order. In this case, $(1 + 2) * 3$ would evaluate to 9 because the parentheses have a higher order of precedence than multiplication does. Table 7-9 defines the order of operator precedence for Python.

TABLE 7-9 Python Operator Precedence
Operator Description

() You use parentheses to group expressions and to override the default precedence so that you can force an operation of lower precedence (such as addition) to take precedence over an operation of higher precedence (such as multiplication).

** Exponentiation raises the value of the left operand to the power of the right operand.

~ + Unary operators interact with a single variable or expression.

* / % // Multiply, divide, modulo, and floor division.

+ Addition and subtraction.

>> << Right and left bitwise shift.

& Bitwise AND.

^ | Bitwise exclusive OR and standard OR.

<= < > >= Comparison operators.

== != Equality operators.

= %= /= //= Assignment operators.

-= += *= **=

**Is is not Identity operators.
In not in Membership operators.
not or and Logical operators.
Creating and Using Functions**

To manage information properly, you need to organize the tools used to perform the required tasks. Each line of code that you create performs a specific task, and you combine these lines of code to achieve a desired result. Sometimes you need to repeat the instructions with different data, and in some cases your code becomes so long that keeping track of what each part does is hard. Functions serve as organization tools that keep your code neat and tidy. In addition, functions make it easy to reuse the inst

**.
the chapter tells you all about functions. More important, in this section you start creating your first serious applications in the same way that professional developers do.**

Viewing functions as code packages

_armed with storage boxes and soon you have everything in the closet in neatly organized boxes. The shoes go in one box, games in another, and old

**.
you want in the closet without fear of injury. Functions are just like that: They take messy code and place it in packages that make it easy to see what you have and understand how it works.**

**.
necessary, but when you boil down all that text, it comes down to a single idea: Functions provide a means of packaging code to make it easy to find and access. If you can think of functions as organizers, you find that working with them is much easier. For example, you can avoid the problem that many developers have of stuffing the wrong items in a function. All your functions will have a single purpose, just like those storage boxes in the closet.**

Understanding code reusability

You go to your closet, take out new pants and shirt, remove the labels, and put them on. At the end of the day, you take everything off and throw it in

take the clothes off, wash them, and then put them back into the closet for reuse. Functions are reusable, too. No one wants to keep repeating the same task; it becomes monotonous and boring. When you create a function, you define a package of code that you can use over and over to perform the same task. All you need to do is tell the computer to perform a specific task by telling it which function to use. The computer faithfully executes each instruction in the function absolutely every time you ask it to do so.

When you work with functions, the code that needs services from the function is named the *caller*, and it calls upon the function to perform tasks for it. Much of the information you see about functions refers to the caller. The caller must supply information to the function, and the function returns information to the caller.

At one time, computer programmers had to keep reinventing the same code.

though, and the concept has evolved over the years until functions have become quite flexible. You can make functions do anything you want. Code reusability is a necessary part of applications to

Reduce
development time
Reduce programmer error
Increase
application reliability
Allow entire
programmer
Make code
groups to benefit from the work of one
easier to understand
Improve

.
**In fact,
applications**

.
**through the
reusability
reusability,**

.
**and 1s into
application efficiency**

**functions do a whole list of things for in the form of reusability. As you
work examples in this book, you see how
makes your life significantly easier. If not for**

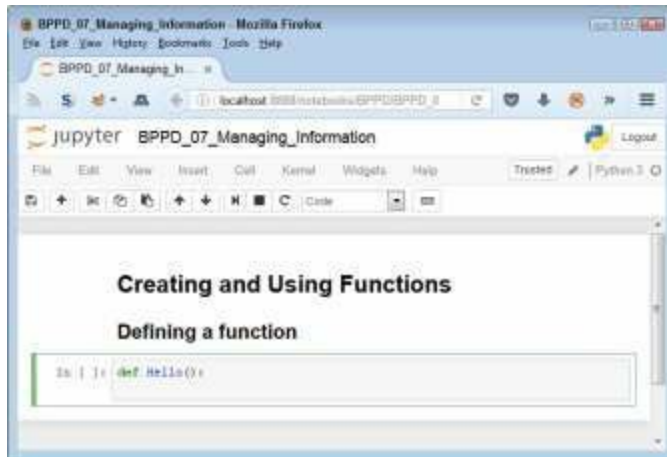
.the computer by hand.

Defining a function
.

**things fast and easy for you. The following steps show you the process of
creating a function that you can later access: 1. Create a new notebook in
Notebook.**

**The book uses the filename BPPD_07_Managing_Information.ipynb,
which is where you find all the source code for this chapter. See the
Introduction for information on using the downloadable source.**

Type def Hello(): and press Enter.



This step tells Python to define a function named Hello. The parentheses are important because they define any requirements for using the function. (There

requirements in this

in which people will access the function. Notice that the insertion pointer is now indented, as shown in Figure 7-1. This indentation is a reminder that you must give the function a task to perform.

FIGURE 7-1:

Define the name of your function.

Type

and press Enter.

You should notice something special, as shown in Figure 7-2. The insertion



FIGURE 7-2:

pointer is still indented because Notebook is waiting for you to provide the next step in the function.

Notebook awaits your next instruction.

Click Run Cell.

The function is now complete.

Even though this is a really simple function, it demonstrates the pattern you use when creating any Python function. You define a name, provide any requirements for using the function (none in this case), and provide a series of steps for using the function. A function ends when the insertion point is at the left side or you move to the next cell.

Accessing functions

After you define a function, you probably want to use it to perform useful work. Of course, this means knowing how to access the function. In the previous section, you create a new function named `Hello()`. To access this function, you type `Hello()` and click Run Cell. Figure 7-3 shows the output you see when you execute this function.

FIGURE 7-3:



Whenever you type the .the output the function provides.

Every function you create will provide a similar pattern of usage. You type the function name, an open parenthesis, any required input, and a close parenthesis; then you press Enter. In this case, you have no input, so all you type is `Hello()`. As the chapter progresses, you see other examples for which input is required.

Sending information to functions

to keep typing that long string every time you want to say `Hello()`.

o quite limited because you can use it to say only one thing. Functions should be flexible and allow you to do more than just one thing. Otherwise, you end up writing a lot of functions that vary by the data they use rather than the functionality they provide. Using arguments

helps

you create functions that are flexible and can use a wealth of data.

Understanding arguments

The term *argument*

.

the function; it means that you supply information to the function to use in processing a request. Perhaps a better word for it would be input, but the term *input* has been used for so many other purposes that developers decided to use something a bit different: argument. Although the purpose of an argument might not be clear from its name, understanding what it does is relatively straightforward. An argument makes it possible for you to send data to the function so that the function can use it when performing a task. Using arguments makes your function more flexible.

The `Hello()` function is currently inflexible because it prints just one string. Adding an argument to the function can make it a lot more flexible because you can send strings to the function to say anything you want. To see how arguments work, create a new function in the notebook. This version of `Hello()`, `Hello2()`, requires an argument:
`def Hello2( Greeting ):`

```
print(Greeting)
```

```
print(Greeting)
```

```
4:
```

You must supply an argument or you get an error message.

```
message. 5:
```

Use

`Hello2()` to print

any

message you

desire.

Notice

that the

parentheses are no longer empty. They contain a word, `Greeting`, which is the argument for `Hello2()`. The `Greeting` argument is actually a variable that you can pass to `print()` in order to see it onscreen.

Sending required arguments

You have a new function, `Hello2()`. This function requires

.w

`Hello2()` and click **Run Cell**. You see an error message, as shown in Figure 7-4, telling you that `Hello2()` requires an argument.



```

Sending required arguments

In [4]: Hello2()

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-4-8a5ee0424a8b> in <module>()
----> 1 Hello2()

TypeError: Hello2() missing 1 required positional argument: 'Greeting'

```

Not only does Python tell you that the argument is missing, it tells you the name of the argument as well. Creating a function the way you have done so far means that you must supply an argument. Type

.

function.

.

and

click **Run Cell**. This time, you see the expected output. However, you still

.know

whether

`Hello2()` is flexible enough to print multiple messages. Type

.

and click **Run Cell**. You see

the expected output again, as shown in Figure 7-5, so

`Hello2()` is indeed an improvement over `Hello()`.

```
In [5]: Hello2("This is an interesting function.")
This is an interesting function.

In [6]: Hello2("Another message...")
Another message...
```

You might easily to assume that Greeting will accept only a string from the tests you have performed so far. Type `Hello2(1234)`, click Run Cell, and you see 1234 as the output. Likewise, type `Hello2(5 + 5)` and click Run Cell. This time you see the result of the expression, which is 10.

Sending arguments by keyword

As your functions become more complex and the methods to use them do as well, you may want to provide a little more control over precisely how you call the function and provide arguments to it. Up until now, you have

positional arguments, which means that you have supplied values in the order in which they appear in the argument list for the function definition. However, Python also has a method for sending arguments by keyword. In this case, you supply the name of the argument followed by an equals sign (=) and the argument value. To see how this works, type the following function in the notebook:

```
def AddIt(Value1, Value2):
```

```
    print(Value1, " + ", Value2, " = ", (Value1 + Value2))
```

Notice that the `print()` function argument includes a list of items to print and that those items are separated by commas. In addition, the arguments are of different types. Python makes it easy to mix and match arguments in this manner.

Time to test `AddIt()`. Of course, you want to try the function using positional arguments first, so type `AddIt(2, 3)` and click Run Cell. You see the expected output of `2 + 3 = 5`. Now type `AddIt(Value2 = 3, Value1 = 2)` and click Run Cell. Again, you receive the output `2 + 3 = 5` even though the position of the arguments has been reversed.

Giving function arguments a default value

Whether you make the call using positional arguments or keyword arguments, the functions to this point have required that you supply a value. Sometimes a function can use default values when a common value is available. Default values make the function easier to use and less likely

default value, you simply follow the argument name with an equals sign and the default value. To see how this works, type the following function in the notebook:

```
def Hello3(Greeting = "No Value Supplied"):
```

```
    print(Greeting)
```

This is yet another version of the original Hello() and updated Hello2() functions, but Hello3() automatically compensates for individuals who .Hello3() and press Enter to see for yourself. Type .to see a normal response. Lest you think the function is now unable to use other kinds of data, type

Giving function arguments a default value

```
In [12]: def Hello3(Greeting = "No Value Supplied"):  
        print(Greeting)  
  
In [13]: Hello3()  
Hello3("This is a string")  
Hello3(0)  
Hello3(2 + 7)  
No Value Supplied  
This is a string  
5  
9
```

FIGURE 7-6:

Hello3(5) and press Enter; then Hello3(2 + 7) and press Enter. Figure 7-6 shows the output from all these tests.

Supply default arguments when possible to make your functions easier to use.

Creating functions with a variable number of arguments

In most cases, you know precisely how many arguments to provide with your function. It pays to work toward this goal whenever you can because functions with a fixed number of arguments are easier to

how many arguments the function will receive at the outset. For example, when you create a Python application that works at the

command line, the user might provide no arguments, the maximum number of arguments (assuming there is more than one), or any number of arguments in between.

Fortunately, Python provides a technique for sending a variable number of arguments to a function. You simply create an argument that has an asterisk in front of it, such as `*VarArgs`. The usual technique is to provide a second argument that contains the number of arguments passed as an input. Here is an example of a function that can print a variable number

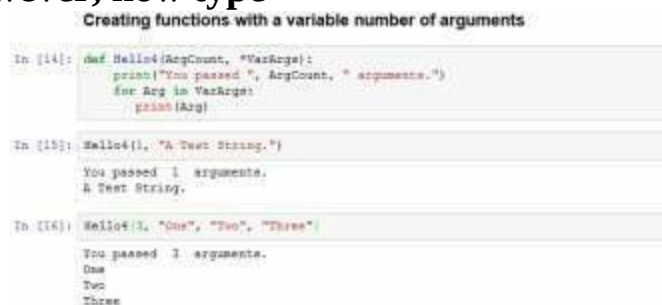
.completely now

.before.)

```
def Hello4(ArgCount, *VarArgs):  
    print("You passed ", ArgCount, " arguments.")  
    for Arg in VarArgs: print(Arg)
```

This example uses something called a for loop. You meet this structure in Chapter 9. For now, all you really need to know is that it takes the arguments out of `VarArgs` one at a time, places the individual argument into `Arg`, and then prints `Arg` using `print()`. What should interest you most is seeing how a variable number of arguments can work.

After you type the function into the notebook, type
.st.and click Run Cell. You should see the number of arguments and the test string as output
.nothing too exiting there. However, now type



The screenshot shows a Jupyter Notebook interface with the title "Creating functions with a variable number of arguments". It contains three code cells. The first cell defines the `Hello4` function. The second cell calls `Hello4(1, "A Test String.")` and produces the output "You passed 1 arguments." and "A Test String.". The third cell calls `Hello4(3, "One", "Two", "Three")` and produces the output "You passed 3 arguments.", "One", "Two", and "Three".

```
Creating functions with a variable number of arguments  
In [14]: def Hello4(ArgCount, *VarArgs):  
          print("You passed ", ArgCount, " arguments.")  
          for Arg in VarArgs:  
              print(Arg)  
  
In [15]: Hello4(1, "A Test String.")  
You passed 1 arguments.  
A Test String.  
  
In [16]: Hello4(3, "One", "Two", "Three")  
You passed 3 arguments.  
One  
Two  
Three
```

.and click Run Cell. As shown

in Figure 7-7, the function handles the variable number of arguments without any problem at all.

FIGURE 7-7:

**Variable argument functions can make your applications more flexible.
Returning information from functions**

Functions can display data directly or they can return the data to the caller so that the caller can do something more with it. In some cases, a function displays data directly as well as returns data to the caller, but more commonly, a function either displays the data directly or returns it to the caller. Just how functions work depends on the kind of task the function is supposed to perform. For example, a function that performs a math-related task is more likely to return the data to the caller than certain other functions.

To return data to a caller, a function needs to include the keyword **return**, followed by the data to return. You have no limit on what you can return to a caller. Here are some types of data that you commonly see returned by a function to a caller:

Any value is 1 or 2.5;
values, such

The content value. The variable.

Many
example,
you can
calculation, **_Expressions:** acceptable. You can return numbers, such as
as **True** or **False**.
of any variable works just as well as a direct caller receives whatever
data is stored in the
developers use expressions as a shortcut. For

simply return $A + B$ rather than perform the place the result in a
variable, and then return the variable to the caller. Using the expression
is faster and accomplishes
the same task.

Results from other functions: You can actually return data from another
function as part of the return of your function.

.notebook:

```
def DoAdd(Value1, Value2):  
return Value1 + Value2
```

This function accepts two values as input and then returns the sum of those two values. Yes, you could probably perform this task without using a function, but this is how many functions start. To test this function, type

and click Run Cell. You see the output shown in Figure 7-8.

Returning information from functions

```
In [17]: def DoAdd(Value1, Value2):  
         return Value1 + Value2
```

```
In [18]: print("The sum of 3 + 4 is ", DoAdd(3,4))
```

The sum of 3 + 4 is 7

FIGURE 7-8:

Return values can make your functions even more useful.

Comparing function output

You use functions with return values in a number of ways. For example, the previous section of this chapter shows how you can use functions to provide input for another function. You use functions to perform all sorts of tasks. One of the ways to use functions is for comparison purposes. You can actually create expressions from them that define a logical output.

To see how this might work, use the DoAdd() function from the previous section. Type

.

and click Run Cell. You see the truth value of the statement that 3 + 4 equals 2 + 5, as shown in Figure 7-9. The point is that functions need not provide just one use or that you view them in just one way. Functions can make your code quite versatile and flexible.

Comparing function output

```
In [19]: print("3 + 4 equals 2 + 5 is ", (DoAdd(3,4)==DoAdd(2,5)))
```

3 + 4 equals 2 + 5 is: True

FIGURE 7-9:

Use your functions to perform a wide variety of tasks.

Getting User Input

Very few applications exist in their own world
that is, apart from the

user. In fact, most applications interact with users in a major way because computers are designed to serve user needs. To interact with a user, an application must provide some means of obtaining user input. Fortunately, the most commonly used technique for obtaining input is also relatively easy to implement. You simply use the `input()` function to do it.

FIGURE 7-10:

Provide a username and see a greeting as output.

Provide a username and see a greeting as output.

11:

Data

conversion changes

the input

type to

whatever you need, but could cause
errors.

The `input()` function

always

outputs a string.

Even if a

user types a number, the output from the

`input()`

function is a string. This means that if you are expecting a number, you need to convert it after receiving the input. The `input()` function also lets you provide a string prompt. This prompt is displayed to tell the user what to provide in the way of information. The following example shows how to use the `input()` function in a simple way:

```
Name = input("Tell me your name: ")  
print("Hello ", Name)
```

In this case, the `input()` function asks the user for a name. After the user types a name and presses Enter, the example outputs a customized greeting to the user. Try running this example. Figure 7-10 shows typical results when you input John as the username.

Getting User Input

```
In [20]: Name = input("Tell me your name: ")
        print("Hello ", Name)

Tell me your name: John
Hello John
```

You can use `input()` for other kinds of data; all you need is the correct conversion function. For example, the code in the following example provides one technique for performing such a conversion, as shown here:

```
ANumber = float(input("Type a number: "))
using a value such as 5.5, you obtain the desired result.
```

```
print("You typed: ", ANumber)
```

attempt to type something other than a number, you get an error message, as shown in Figure 7-11. Chapter 10 helps you understand how to detect and fix errors before they cause a system crash.

```
In [21]: ANumber = float(input("Type a number: "))
        print("You typed: ", ANumber)

Type a number: Hello

-----
ValueError                                Traceback (most recent call last)
<ipython-input-21-5b050a97d4dc> in <module> ()
----> 1 ANumber = float(input("Type a number: "))
      2 print("You typed: ", ANumber)

ValueError: could not convert string to float: 'Hello'
```

When you run this example, the application asks for a numeric input. The call to `float()` converts the input to a number.

After the conversion , print()
outputs
the result. When you run the
example

8 Making Decisions

T

he ability to make a decision, to take one path or another, is an essential element of performing useful work. Math gives the computer the capability to obtain useful information. Decisions enable you to do something with the

information after obtaining it. Without the capability to make decisions, a computer would be useless. So any language you use will include the capability to make decisions in some manner. This chapter explores the techniques that Python uses to make decisions. You can find the downloadable source code for this chapter in the

Introduction.

Think through the process you use when making a decision. You obtain the actual value of something, compare it to a desired value, and then act .you compare the red light to the desired green light, decide that the light

process they use because they use it so many times every day. Decision making comes naturally to humans, but computers must perform the following tasks every time:

Obtain the actual or current value of something.

Compare the actual or current value to a desired value.

Perform an action that corresponds to the desired outcome of the comparison.

Making Simple Decisions by Using the if Statement

The if statement is the easiest method for making a decision in Python. It simply states that if something is true, Python should perform the steps that follow. The following sections tell you how you can use the if statement to make decisions of various sorts in Python. You may be

surprised at what this simple statement can do for you.

Understanding the if statement

You use if statements regularly in everyday life. For example, you may say

Python if statement is a little less verbose, but it follows precisely the same pattern. Say you create a variable, TestMe, and place a value of 6 in it, like this:

```
TestMe = 6
```

You can then ask the computer to check for a value of 6 in TestMe, like this:

```
if TestMe == 6:  
    print("TestMe does equal 6!")
```

Every Python if statement begins, oddly enough, with the word *if*. When Python sees if, it knows that you want it to make a decision. After the word *if* comes a condition. A *condition* simply states what sort of comparison you

want Python to make. In this case, you want Python to determine whether TestMe contains the value 6.

Notice that the condition uses the relational equality operator, `==`, and not the assignment operator, `=`. A common mistake that developers make is to use the assignment operator rather than the equality operator. You can see a list of relational operators in Chapter 7.

-

look for additional conditions on which to base its decision. After the colon come any tasks you want Python to perform. In this case, Python prints a statement saying that TestMe is equal to 6.

Using the if statement in an application

You can use the if statement in a number of ways in Python. However, you immediately need to know about three common ways to use it: Use a single

.condition to execute a single statement when the condition is

true. Use a single

when the

condition to execute multiple statements condition

Combine

.

and execute `is true`. condition is

The following

provide you

.

additional

multiple conditions into a single decision one or more statements when the combined true.

sections explore these three possibilities and with examples of their use.

You see

examples of how to use the if statement

.decisions.

Working with relational operators

A *relational operator* determines how a value on the left side of an expression compares to the value on the right side of an expression. After it makes the determination, it outputs a value of true or false that reflects the truth value of the expression. For example, `6 == 6` is true, while `5 == 6` is false. Table 7-3 in Chapter 7 lists the relational operators. The following steps show how to create and use an if statement.

Open a new notebook.

You can also use the downloadable source file,

`BPPD_08_Making_Decisions`.

`ipynb`.

Type `TestMe = 6` and press Enter.

This step assigns a value of 6 to TestMe. Notice that it uses the assignment operator and not the equality operator.
Type `if TestMe == 6:` and press Enter.

This step creates an if statement that tests the value of TestMe by using the equality operator. You should notice two features of Notebook at this point:

The word *if* is highlighted in a different color than the rest of the statement.

The next line is automatically indented.

Type
and press Enter.

Notice that Python doe

.

the next line. The word *print*

.

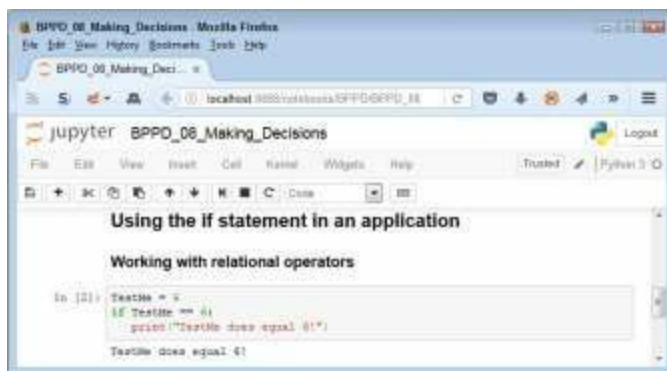
function name. In addition, the text appears in another color to show you

.to see how Python works.

Click Run Cell.

Notebook executes the if statement, as shown in Figure 8-1. Notice that the output is in yet another color. Because TestMe contains a value of 6, the if statement works as expected.

FIGURE 8-1:



Simple if statements can help

Performing multiple tasks

Sometimes you want to perform more than one task after making a decision. Python relies on

indentation to determine when to stop executing tasks as part of an if statement. As long as the next line

.

statement. When the next line is outdented, it becomes the first line of code outside the if block. A *code block* consists of a statement and the tasks associated with that statement. The same term is used no matter what kind

an if statement that is part of a code block. The following steps show how to use indentation to execute multiple steps as part of an if statement.

1. Type the following code into the notebook

pressing Enter after each line:

```
TestMe = 6
```

```
if TestMe == 6: print("TestMe does equal 6!")
```

```
print("All done!")
```

FIGURE 8-2:

Each line you type is part of the current if statement code block.

A code
block can contain
multiple lines of
code

.

one for
each task.

Notice that the shell
continues to indent
lines as

long as you continue to type code. When working in the shell, you create a block by typing one line of code after another. If you press Enter twice

in a row without entering any text, the code block is ended, and Python executes the entire code block at one time. Of course, when working in Notebook, you must click Run Cell to execute the code within that cell. 2. Click Run Cell.

Python executes the entire code block. You see the output shown in Figure 8-2.



```
Performing multiple tasks
In [3]: TestMe = 6
        if TestMe == 6:
            print('TestMe does equal 6!')
            print('All done!')

TestMe does equal 6!
All done!
```

Making multiple comparisons by using logical operators

So far, the examples have all shown a single comparison. Real life often requires that you make multiple comparisons to account for multiple requirements. For example, when baking cookies, if .take the cookies out of the oven.

To make multiple comparisons, you create multiple conditions by using relational operators and combine them by using logical operators (see Table 7-4 in Chapter 7). A

logical operator describes how to combine conditions. For example, you might say `x == 6` and `y == 7` as two conditions for performing one or more tasks. The `and` keyword is a logical operator that states that both conditions must be true.

One of the most common uses for making multiple comparisons is to determine when a value is within a certain range. In fact, *range checking*, the act of determining whether data is between two values, is an important part of making your application secure and user friendly. The following steps help you see how to perform this task. In this case, you create a file so that you can run the application multiple times.

1. Type the following code into the notebook pressing Enter after each line:

FIGURE 8-3:

The application verifies the value is in the right range and outputs a message.

```
Value = int(input("Type a number between 1 and 10: "))
```

The

```
if (Value > 0) and (Value <= 10):  
    print("You typed: ", Value)
```

example

begins by

obtaining

an input

value. You have no

idea what

the user

has typed other than

.

value of

some sort. The use of the int()

function

means that the user

must type a whole

number (one without a

decimal

portion).

Otherwise, the

application will raise an *exception*(an error indication; Chapter 10 describes exceptions). This first check ensures that the input is at least of the correct type.

The if statement contains two conditions. The first states that Value must be greater than 0. You could also present this condition as Value >= 1.

The second condition

states that Value must be less than or equal to 10. Only when Value meets both of these conditions will the if statement succeed and print the value the user typed.

Click Run Cell.

Python prompts you to type a number between 1 and 10.

Type 5 and press Enter.

The application determines that the number is in the right range and outputs the message shown in Figure 8-3.

Making multiple comparisons using logical operators

```
In [4]: Value = int(input("Type a number between 1 and 10: "))
if (Value > 0) and (Value <= 10):
    print("You typed: ", Value)
```

Type a number between 1 and 10: 5
You typed: 5

Select the cell again. Repeat Steps

2 and 3, but type 22 instead of 5.

number is in the wrong range. Whenever you type a value

Note that the input value updates by one. Each time you execute a cell, the input value will change. Given that the input value is at 4 in Figure 8-3, you now see In [5]: in the Notebook margin.

Making multiple comparisons using logical operators

```
In [5]: Value = int(input("Type a number between 1 and 10: "))
if (Value > 0) and (Value <= 10):
    print("You typed: ", Value)
```

Type a number between 1 and 10: 5.5

ValueError Traceback (most recent call last)
 <ipython-input-5-2a1fa76bbdd9> in <module>()
----> 1 Value = int(input("Type a number between 1 and 10: "))
 2 if (Value > 0) and (Value <= 10):
 3 print("You typed: ", Value)

ValueError: invalid literal for int() with base 10: '5.5'

Select the

3, but type 5.5 instead of 5.

Python displays the error message shown in Figure 8-4. Even though you

may think of 5.5 and 5 as both being numbers, Python

cell again. sees the first number as a floating-point value and the Repeat second as an integer. (Note also that the input value is now Steps 2 and at 6.)

FIGURE 8-4:

Typing the wrong type of information results in an error message.

Repeat Steps 2 and 3, but type Hello instead of 5.

.differentiate between types of wrong input. It knows only that the input type is incorrect and therefore unusable.

The best applications use various kinds of range checking to ensure that the application behaves in a predictable manner. The more predictable an application becomes, the less the user thinks about the application and the more time the user spends on performing useful work.

Productive users tend to be a lot happier than those who constantly fight with their applications.

Choosing Alternatives by Using the if. . .else Statement

Many of the decisions you make in an application fall into a category of choosing one of two options based on conditions. For example, when looking at a signal light, you choose one of two options: press on the brake to stop or press the accelerator to continue. The option you choose depends on the conditions. A green light signals that you can continue on through the light; a red light tells you to stop. The following sections describe how Python makes choosing between two alternatives possible.

Understanding the if. . .else statement

With Python, you choose one of two alternatives by using the else clause of the if statement. A *clause* is an addition to a code block that modifies the way in which it works. Most code blocks support multiple clauses. In this case, the else clause enables you to perform an alternative task, which increases the usefulness of the if statement. Most developers refer to the form of the if statement that has the else clause included as the if...else statement, with the ellipsis implying that something happens between if and else.

Sometimes developers encounter problems with the if...else statement because they forget that the else clause always executes when the consequences of always executing a set of tasks when the conditions are false. Sometimes doing so can lead to unintended consequences.

Using the if. .else statement in an application

The example in the previous section is a little less helpful than it could be

entering data of the wrong type produces an error message, but entering the correct type of data outside the range tells the user nothing. In this example, you discover the means for correcting this problem by using an else clause. The following steps demonstrate just one reason to provide an alternative action when the condition for an if statement is false:

Type the following code into the notebook

pressing Enter after each line:

```
Value = int(input("Type a number between 1 and 10: "))
```

```
if (Value > 0) and (Value <= 10):
```

```
    print("You typed: ", Value)
```

```
else:
```

```
    print("The value you typed is incorrect!")
```

As before, the example obtains input from the user and then determines whether that input is in the correct range. However, in this case, the else clause provides an alternative output message when the user enters data outside the desired range.

Notice that the else clause ends with a colon, just as the if statement does. Most clauses that you use with Python statements have a colon associated with them so that Python knows when the clause has ended. If you receive a coding error for your application, make sure that you check for the presence of the colon as needed.

Click Run Cell.

Python prompts you to type a number between 1 and 10.

Type 5 and press Enter.

The application determines that the number is in the right range and outputs the message shown previously in Figure 8-3.

.

**Repeat
Steps 2
and 3, but type 22
instead of 5.**

**This time the
application outputs the error message shown in Figure 8-5. The user
now knows that the input is outside the desired range and knows to try
entering it again.**

Using the if...else statement in an application

```
In [8]: Value = int(input("Type a number between 1 and 10: "))
if (Value > 0) and (Value <= 10):
    print("You typed: ", Value)
else:
    print("The value you typed is incorrect!")

Type a number between 1 and 10: 5
You typed: 5
```



```
In [9]: Value = int(input("Type a number between 1 and 10: "))
if (Value > 0) and (Value <= 10):
    print("You typed: ", Value)
else:
    print("The value you typed is incorrect!")

Type a number between 1 and 10: 22
The value you typed is incorrect!
```

Using the if . . . elif statement in an application

**You go to a
restaurant and look at the
menu. The restaurant offers
eggs,
pancakes, waffles,
and
oatmeal for
breakfast. After you choose
one of the items, the server
brings it
to you.**

**Creating a menu
selection requires something like an if...else statement, but with a little
extra oomph. In this case, you use the elif clause to create another set of
conditions. The elif clause is a combination of the else clause and a
separate if statement. The following steps describe how to use the if...elif
statement to create a menu.**

Type the following code into the notebook

pressing Enter after each line:

```
print("1. Red")
print("2. Orange")
print("3. Yellow")
print("4. Green")
print("5. Blue")
print("6. Purple") print("You chose Red!")
Choice = int(input("Select your favorite color: "))
elif (Choice == 2):
if (Choice == 1):
print("You chose Orange!")
elif (Choice == 3):
print("You chose Yellow!")
elif (Choice == 4):
print("You chose Green!")
elif (Choice == 5):
print("You chose Blue!") elif (Choice == 6):
print("You chose Purple!")
else:
print("You made an invalid choice!")
```

The example begins by displaying a menu. The user sees a list of choices for the application. It then asks the user to make a selection, which it places anything other than a number.

After the user makes a choice, the application looks for it in the list of potential values. In each case, Choice is compared against a particular value to create a condition for that value. When the user types 1, the application outputs the message "You chose Red!". If none of the options is correct, the else clause is executed by default to tell the user that the input choice is invalid.

Click Run Cell.

Python displays the menu. The application asks you to select your favorite color.

Type 1 and press Enter.

The application displays the appropriate output message, as shown in Figure 8-6.

Repeat Steps 3 and 4, but type 5 instead of 1.

The application displays a different output message the one associated with the requested color.

Repeat Steps 3 and 4, but type 8 instead of 1.

The application tells you that you made an invalid choice.

Using the if...elif statement in an application

```
In [40]: print("1. Red")
        print("2. Orange")
        print("3. Yellow")
        print("4. Green")
        print("5. Blue")
        print("6. Purple")
        Choice = int(input("Select your favorite color: "))
        if (Choice == 1):
            print("You chose Red!")
        elif (Choice == 2):
            print("You chose Orange!")
        elif (Choice == 3):
            print("You chose Yellow!")
        elif (Choice == 4):
            print("You chose Green!")
        elif (Choice == 5):
            print("You chose Blue!")
        elif (Choice == 6):
            print("You chose Purple!")
        else:
            print("You made an invalid choice!")

1. Red
2. Orange
3. Yellow
4. Green
5. Blue
6. Purple
Select your favorite color: 1
You chose Red!
```

inputs.

Repeat Steps 3 and 4, but type Red instead of 1.

The application displays the expected error message, as shown in Figure 8-7.

Any application you create should be able to detect errors and incorrect

Chapter 10 shows you how t

FIGURE 8-6:

Menus let you choose one option from a list of options.

FIGURE 8-7:

Every application you create should include some means of detecting errant input.

```

In [13]: print("1. Red")
          print("2. Orange")
          print("3. Yellow")
          print("4. Green")
          print("5. Blue")
          print("6. Purple")
          Choice = int(input("Select your favorite color: "))
          if (Choice == 1):
              print("You chose Red!")
          elif (Choice == 2):
              print("You chose Orange!")
          elif (Choice == 3):
              print("You chose Yellow!")
          elif (Choice == 4):
              print("You chose Green!")
          elif (Choice == 5):
              print("You chose Blue!")
          elif (Choice == 6):
              print("You chose Purple!")
          else:
              print("You made an invalid choice!")

1. Red
2. Orange
3. Yellow
4. Green
5. Blue
6. Purple
Select your favorite color: Red

=====
ValueError                                Traceback (most recent call last)
<ipython-input-13-7f8hofafafes> in <module>()
      5 print("5. Blue")
      6 print("6. Purple")
----> 7 Choice = int(input("Select your favorite color: "))
      8 if (Choice == 1):
      9     print("You chose Red!")

ValueError: invalid literal for int() with base 10: 'Red'

```

NO SWITCH STATEMENT?

other languages, you might notice

is no need to worry about it with Python). Developers commonly use the switch statement in other languages to create menu-based applications. The if...elif statement is generally used for the same purpose in Python.

same functionality as a switch statement because it

purposes. As a result, some dev

dictionary functionality to stand in for the switch statement. Chapter 14 describes how to work with dictionaries.

Using Nested Decision Statements

The decision-making process often happens in levels. For example, when you go to the restaurant and choose eggs for breakfast, you have made a first-level decision. Now the server asks you what type of toast you want

pancakes, so the selection of toast becomes a second-level decision. When the breakfast arrives, you decide whether you want to use jelly on your toast. This is a third-level decision. If you had selected a kind of toast that

.

at all. This process of making decisions in levels, with each level reliant on the decision made at the previous level, is called *nesting*. Developers often use nesting techniques to create applications that can make complex decisions based on various inputs. The following sections describe several kinds of nesting you can use within Python to make complex decisions.

Using multiple if or if...else statements

The most commonly used multiple selection technique is a combination of if and if...else statements. This form of selection is often called a *selection tree* because of its resemblance to the branches of a tree. In this case, you follow a particular path to obtain a desired result. The following steps show how to create a selection tree:

Type the following code into the notebook

pressing Enter after each line:

```
One = int(input("Type a number between 1 and 10: "))
```

```
Two = int(input("Type a number between 1 and 10: "))
```

```
if (One >= 1) and (One <= 10):
```

```
if (Two >= 1) and (Two <= 10):
```

```
print("Your secret number is: ", One * Two)
```

```
else:
```

```
print("Incorrect second value!") else:
```

```
print("Incorrect first value!")
```

.else statement in an

.

that the indentation is different. The second if...else statement is indented within the first if...else statement. The indentation tells Python that this is a second-level statement.

Click Run Cell.

You see a Python Shell window open with a prompt to type a number between 1 and 10.

Type 5 and press Enter.

The shell asks for another number between 1 and 10.

Type 2 and press Enter.

**You see the
of the two
output, as
Figure 8-8.**

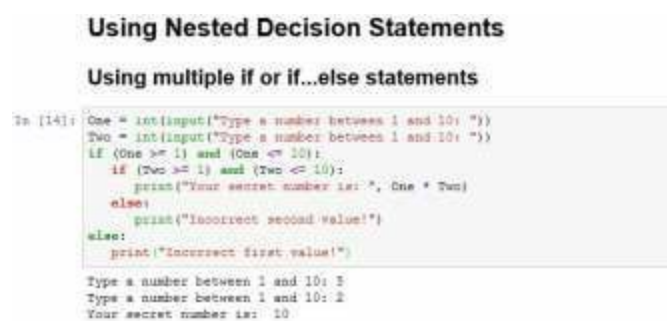


FIGURE 8-8:

Adding multiple levels lets you perform tasks with greater complexity.

combination numbers as shown in

This example has the same input features as the previous if. . .else example.

requested range, you see an error message. The error message is tailored for either the first or second input value so that the user knows which value was incorrect.

Providing specific error messages is always useful because users tend to become confused and frustrated otherwise. In addition, a specific error message helps you find errors in your application much faster.

Combining other types of decisions

You can use any combination of if, if...else, and if...elif statements to produce a desired outcome. You can nest the code blocks as many levels

deep as needed to perform the required checks. For example, Listing 8-1 shows what you might accomplish for a breakfast menu.

[REDACTED]

1:

```
print("1. Eggs") print("2. Pancakes") print("3. Waffles") print("4.
Oatmeal") MainChoice = int(input("Choose a breakfast item: ")) if
(MainChoice == 2): Meal = "Pancakes" elif (MainChoice == 3): Meal =
"Waffles" if (MainChoice == 1): print("1. Wheat Toast") print("2. Sour
Dough") print("3. Rye Toast") print("4. Pancakes") Bread =
int(input("Choose a type of bread: ")) if (Bread == 1):
```

```
print("You chose eggs with wheat toast.") elif (Bread == 2):
print("You chose eggs with sour dough.") elif (Bread == 3):
print("You chose eggs with rye toast.") elif (Bread == 4):
print("You chose eggs with pancakes.") else:
print("We have eggs, but not that kind of bread.") elif (MainChoice ==
2) or (MainChoice == 3):
print("1. Syrup") print("2. Strawberries") print("3. Powdered Sugar")
Topping = int(input("Choose a topping: "))
if (Topping == 1):
print ("You chose " + Meal + " with syrup.") elif (Topping == 2):
print ("You chose " + Meal + " with strawberries.") elif (Topping == 3):
print ("You chose " + Meal + " with powdered sugar.") else:
print ("We have " + Meal + ", but not that topping.") elif (MainChoice
== 4):
print("You chose oatmeal.") else: print("We don't serve that breakfast
item!")
```

This example has some interesting features. For one thing, you might assume that an if...elif statement always requires an else clause. This

.

if...elif statement to ensure that Meal contains the correct value, but you have no other options to consider.

The selection technique is the same as you saw for the previous examples.

A user enters a number in the correct range to obtain a desired result. Three of the selections require a secondary choice, so you see the menu for that

.topping, but you do want a topping for pancakes or waffles.

Notice that this example also combines variables and text in a specific way. Because a topping can apply equally to waffles or pancakes, you need some method for defining precisely which meal is being served as part of the output. The Meal variable that the application defines earlier is used as part of the output after the topping choice is made.

The best way to understand this example is to play with it. Try various menu combinations to see how the application works.

Chapter 9 Performing Repetitive Tasks

A

ll the examples in the book so far have performed a series of steps just one

Many of the tasks that humans perform are repetitious. For example, the doctor might state that you need to exercise more and tell you to do 100 push-ups each day. If you just do one push

-

- ups to do, you can perform the task a specific number of times. Python allows the same sort of repetition by using the for statement.

.For example, consider needing to check a stack of coins for one of extreme rarity. Taking just the first coin from the top, examining it, and deciding

.you must examine each coin in turn, looking for the rare coin. Your stack may contain more than one. Only after you have looked at every coin in the

to perform the task at the outset. You only know the task is done when the stack is gone. Python performs this kind of repetition by using the while statement.

Most programming languages call any sort of repeating sequence of events a *loop*. The idea is to picture the repetition as a circle, with the code going round and round executing tasks until the loop ends. Loops are an essential part of application elements such as menus. In fact, writing most modern applications without using loops would be impossible.

In some cases, you must create loops within loops. For example, to create a multiplication table, you use a loop within a loop. The inner loop calculates the column values and the outer loop moves between rows. You see such

understanding precisely how such things work right now. You can find the downloadable source code for this chapter in the `BPPD_09_Performing_Repetitive_Tasks.ipynb` file, as described in the

Processing Data Using the for Statement

The first looping code block that most developers encounter is the for

language that lacks such a statement. In this case, the loop executes a fixed number of times, and you know the number of times it will execute before the loop even begins. Because everything about a for loop is known at the outset, for loops tend to be the easiest kind of loop to use. However, in order to use one, you need to know how many times to execute the loop. The following sections describe the for loop in greater detail. Understanding the for statement

A for loop begins with a for statement. The for statement describes how to perform the loop. The Python for loop works through a sequence of some series of letters in a string or items within a collection. You can even specify a range of values to
`for Letter in "Howdy!":`

The statement begins with the keyword `for`. The next item is a variable that holds a single element of a sequence. In this case, the variable name is `Letter`. The `in` keyword tells Python that the sequence comes next. In this case, the sequence is the string `"Howdy"`. The for statement always

ends with a colon, just as the decision-making statements described in Chapter 8 do.

Indented under the for statement are the tasks you want performed within the for loop. Python considers every following indented statement part of the code block that composes the for loop. Again, the for loop works just like the decisionmaking statements in Chapter 8.

Creating a basic for loop

The best way to see how a for loop actually works is to create one. In this case, the example uses a string for the sequence. The for loop processes each of the characters in the string in turn until it runs out of characters.

Open a new notebook.

You can also use the downloadable source file, `BPPD_09_Performing_Repetitive_Tasks.ipynb`.

Type the following code into the notebook pressing Enter after each line:

```
LetterNum = 1
for Letter in "Howdy!":
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

The example begins by creating a variable, `LetterNum`, to track the number of letters that have been processed. Every time the loop completes, `LetterNum` is updated by 1.

The for statement works through the sequence of letters in the string "Howdy!". It places each letter, in turn, in `Letter`. The code that follows displays the current `LetterNum` value and its associated character found in `Letter`.

Click Run Cell.

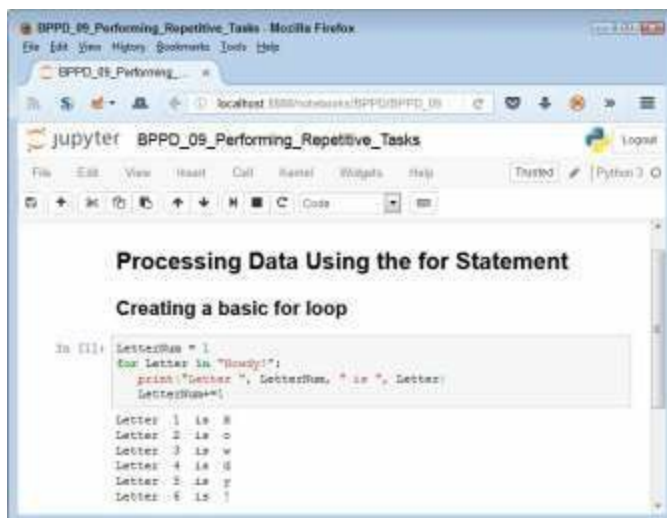
The application displays the letter sequence along with the letter number, as shown in Figure 9-1.

Controlling execution with the break statement

Life is often
exceptions

For
might want
line to
number of

However,
the
runs out of



The screenshot shows a Jupyter Notebook interface in a Mozilla Firefox browser. The notebook title is 'BPPD_05_Performing_Repetitive_Tasks'. The code cell contains a for loop that iterates over the string 'Howdy!'. The output shows the letters of the word 'Howdy!' printed one by one, with the letter number increasing from 1 to 6.

```
In [1]: LetterNum = 1
for Letter in "Howdy!":
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

Letter: 1 is H
Letter: 2 is o
Letter: 3 is w
Letter: 4 is d
Letter: 5 is y
Letter: 6 is !

If the part
about
to the rule.
example, you an assembly produce a
clocks.
at some point, assembly line a needed part.

available, the assembly line must stop in the middle of the processing
cycle.

until
the missing part is restocked.

FIGURE 9-1:

Use the for loop to process the characters in a string one at a time.

Interruptions also occur in computers. You might be streaming data from an online source when a network glitch occurs and breaks the connection; the stream temporarily runs dry, so the application runs out of things to do even though the s

**.The break clause makes breaking out of a loop possible. .code
you surround it with an if statement that defines the**

**condition for issuing a break. The statement might say something like
this: If the stream runs dry, then break out of the loop.**

In this example, you see what happens when the count reaches a certain level when processing a string. The example is a little contrived in the interest of keeping things simple, but it reflects what could happen in the real world when a data element is too long to process (possibly indicating an error condition). 1. Type the following code

into the notebook

.pressing Enter after each line:

Value = input("Type less than 6 characters: ")

LetterNum = 1

for Letter in Value:

print("Letter ", LetterNum, " is ", Letter)

LetterNum+=1 if LetterNum > 6:

print("The string is too long!")

break

break

2:

Long strings are truncated

**to ensure that they remain a certain size. A short
string is
successful ly
processed by**

the
application.

n. 3:

This example builds on the one found in the previous section. However, it lets the user provide a variablelength string. When the string is longer than six characters, the application stops processing it.

The if statement contains the conditional code. When LetterNum is greater than 6, it means that the string is too long. Notice the second level of indentation used for the if statement. In this case, the user sees an error message stating that the string is too long, and then the code executes a break to end the loop.

Click Run Cell.

Python displays a prompt asking for input.

Type
Hello and press
Enter.

The
application lists
each
character in the
string, as shown in shown in 2.

Controlling execution with the break statement

```
In [2]: Value = input("Type less than 6 characters: ")
LetterNum = 1
for Letter in Value:
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
    if LetterNum > 6:
        print("The string is too long!")
        break
```

Type less than 6 characters: Hello
Letter 1 is H
Letter 2 is e
Letter 3 is l
Letter 4 is l
Letter 5 is o

Perform Steps 2 and 3 again, but type I am too long. instead of Hello.

The application displays the expected error message and stops processing the string at character 6, as shown in Figure 9-3.

```
In [3]: Value = input("Type less than 6 characters: ")
LetterNum = 1
for Letter in Value:
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
    if LetterNum > 6:
        print("The string is too long!")
        break

Type less than 6 characters: I am too long.
Letter 1 is I
Letter 2 is a
Letter 3 is m
Letter 4 is t
Letter 5 is o
Letter 6 is o
The string is too long!
```

This example adds *length checking* to your repertoire of application data error checks. Chapter 8 shows how to perform range checks, which ensure that a value meets specific limits. The length check is necessary to ensure tha

fields. In addition, a small input size makes it harder for intruders to perform certain types of hacks on your system, which makes your system more secure.

Controlling execution with the continue statement

want to process certain elements. For example, you might decide that you want to process all the information for every car in a database except brown cars. Perhaps you

particular color of car. The break clause simply ends the loop, so you elements in the sequence.

The break clause alternative that many developers use is the continue clause. As with the break clause, the continue clause appears as part of an if statement. However, processing continues with the next element in the sequence rather than ending completely.

The following steps help you see how the continue clause differs from the break clause. In this case, the code refuses to process the letter w, but will

process every other letter in the alphabet.

Type the following code into the notebook pressing Enter after each line:

```
LetterNum = 1
for Letter in "Howdy!":
    if Letter == "w":
        continue
    print("Encountered w, not processed.")
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

section, earlier in this chapter. However, this example adds an if statement with the continue clause in the if code block. Notice the print() function that is part of the if code block. You never see this string printed because the current loop iteration ends immediately.

Click Run Cell.

Python displays the letter sequence along with the letter number, as shown in Figure 9-4. However, notice the effect of the continue clause

the letter w

FIGURE 9-4:

Use the
clause to
specific

Controlling execution with the continue statement

```
In [4]: LetterNum = 1
for Letter in "Howdy!":
    if Letter == "w":
        continue
    print("Encountered w, not processed.")
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

Letter 1 is H
Letter 2 is o
Letter 3 is d
Letter 4 is y
Letter 5 is !

Controlling

pass clause

continue avoid processing elements.

execution with the

The Python language includes something not commonly found in other languages: a second sort of continue clause. The pass clause works almost the same way as the continue clause does, except that it allows completion of the code in the if code block in which it appears. The following steps use an example that is precisely the same as the one found

.t uses a pass clause instead. 1. Type the following code into the notebook pressing Enter after each line:

```
LetterNum = 1
for Letter in "Howdy!":
    if Letter == "w": pass
    print("Encountered w, not processed.")
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

2. Click Run Cell.

You see a Python Shell window open. The application displays the letter sequence along with the letter number, as shown in Figure 9-5. However, notice the clause

.
processed.
example
that
for the

Controlling execution with the pass clause

```
In [5]: LetterNum = 1
for Letter in "Howdy!":
    if Letter == "w":
        pass
    print("Encountered w, not processed.")
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
```

Letter 1 is H
Letter 2 is o
Encountered w, not processed.
Letter 3 is w
Letter 4 is d
Letter 5 is y
Letter 6 is !

example.

FIGURE 9-5:

effect of the pass the letter w

.

In addition, the
displays the string

.continue clause

Using the pass clause allows for post processing of an unwanted input. The continue clause enables you to silently bypass specific elements in a sequence and avoid executing any additional code for that element. Use the pass clause when you need to perform some sort of post processing on the element, such as logging the element in an error log, displaying a message to the user, or handling the problem element in some other way.

.distinctly different situations.

Controlling execution with the else statement

.else. The else clause makes executing code possible even if you have no elements to process in a sequence. For example, you might need to convey

.

following example does. This example also appears with the downloadable source code as ForElse.py. 1. Type the following code into the notebook

.

pressing Enter after each line:

```
Value = input("Type less than 6 characters: ")
LetterNum = 1
for Letter in Value:
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
else:
    print("The string is blank.")
```

section, earlier in the chapter. However, when a user presses Enter without typing something, the else clause is executed.

Click Run Cell.

Python displays a prompt asking for input.

Type Hello and press Enter.

The application lists each character in the string, as shown previously in Figure 9-2. However, notice that you also see a statement saying that the string is blank, which seems counterintuitive. When using an else clause with a for loop, the else clause always executes. However, if the iterator

ending statement for any for loop. See the article at http://pythonnotes.curious efficiency.org/en/latest/python_concepts/break_else.html for additional details.

Repeat Steps 2 and 3. However, simply press Enter instead of entering any sort of text.

You see the alternative message shown in Figure 9-6 that tells you the string is blank.

FIGURE 9-6:

The else

possible to



```
In [7]: Value = input("Type less than 6 characters: ")
LetterNum = 1
for Letter in Value:
    print("Letter ", LetterNum, " is ", Letter)
    LetterNum+=1
else:
    print("The string is blank.")

Type less than 6 characters:
The string is blank.
```

clause makes it perform tasks

based on an empty sequence.

always signify a simple lack of input. An empty sequence can also signal an application error or other conditions that need to be handled differently from a simple omission of data. Make sure you understand how the

hiding potential error conditions, rather than making them visible so that they can be fixed.

Processing Data by Using the while Statement

much data the application will have to process. Instead of instructing Python to process a static number of items, you use the while statement to tell Python to continue processing items until it runs out of items. This kind of loop is useful when you need to perform tasks such as downloading files of unknown size or streaming data from a source such as a radio station.

an application will process is a good candidate for the while statement, which the following sections describe more fully.

Understanding the while statement

The while statement works with a condition rather than a sequence. The condition states that the while statement should perform a task until the condition is no longer true. For example, imagine a deli with a number of customers standing in front of the counter. The salesperson continues to service customers until no more customers are left in line. The line could

be impossible to know at the outset how many customers will be served. All the salesperson knows is that continuing to serve customers until no more are left is important. Here is how a while statement might look:

```
while Sum < 5:
```

The statement begins with the while keyword. It then adds a condition. In this case, a variable, Sum, must be less than 5 for the loop to continue. Nothing specifies the current value of Sum, nor does the code define how the value of Sum will change. The only thing that is known when Python executes the statement is that Sum must be less than 5 for the loop to continue performing tasks. The statement ends with a colon and the tasks are indented below the statement.

number of times, creating an *endless loop* is possible, meaning that the

loop never ends. For example, say that Sum is set to 0 when the loop begins, and the ending condition is that Sum must be less than 5. If the value of Sum never increases, the loop will continue executing forever (or at least until the computer is shut down). Endless loops can cause all sorts of bizarre problems on systems, such as slowdowns and even computer

the loop to end when using a while loop (contrasted with the for loop, in which the end of the sequence determines the end of the loop). So, when working with the while statement, you must perform three tasks:

Create the environment for the condition (such as setting Sum to 0).
State the condition within the while statement (such as $\text{Sum} < 5$).
Update the condition as needed to ensure that the loop eventually ends (such as adding $\text{Sum} += 1$ to the while code block).

As with the for statement, you can modify the default behavior of the while statement. In fact, you have access to the same four clauses to modify the while statement behavior:

Ends the .break: current loop. Immediately ends processing of the current element.

Ends

completing .continue: processing of the current element after the statements

Provides an conditions

Using the .pass: in the if block.

You can use first example based on the

else: met

**for the loop. alternative processing technique when
.while statement in an application**

the while statement in many ways, but this is straightforward. It simply

displays a count starting and ending condition of a variable named Sum. The following steps help you create and test the example code.

Type the following code into the notebook
pressing Enter after each line:

```
Sum = 0
while Sum < 5:
    print(Sum)
    Sum+=1
```

The example code demonstrates the three tasks you must perform when working with a while loop in a straightforward manner. It begins by setting Sum to 0, which is the first step of setting the condition environment. The condition itself appears as part of the while statement. The end of the while code block accomplishes the third step. Of course, the code displays the current value of Sum before it updates the value of Sum.

A while statement provides flex

.

statement. This example shows a relatively straightforward way to update Sum. However, you can use any update method required to meet the goals of the application. Nothing says that you have to update Sum in a specific manner. In addition, the condition can be as complex as you want it to be. For example, you can track the current value of three or four variables if you want. Of course, the more complex you make the condition, the more likely you are to create an endless loop, so you have a practical limit as to how complex you should make the while loop condition.

Click Run

Python
loop and
sequence

FIGURE 9-7:

Processing Data Using the while Statement

Using the while statement in an application

```
In [5]: Sum = 0
while Sum < 5:
    print(Sum)
    Sum+=1

0
1
2
3
4
```

The simple

sequence of
numbers.

Nesting Loop Statements
Cell.

executes the while displays the numeric shown in Figure 9-7.

while loop displays a

In some cases, you can use either a for loop or a while loop to achieve the same effect. The loop statements work differently, but the effect is the same. In this example, you create a multiplication table generator by nesting a while loop within a for loop. Because you want the output to look nice, you use a little formatting as well. (Chapter 12 gives you the details.)

1. Type the following code into the notebook
pressing Enter after each line:

```
X = 1
Y = 1 print('{:>4}'.format(' '), end= ' ')
for X in range(1, 11):
    print('{:>4}'.format(X), end=' ')
    print()
    for X in range(1,11):
        print('{:>4}'.format(X), end=' ')
        while Y <= 10:
            print('{:>4}'.format(X * Y), end=' ')
            Y+=1 print()
        Y=1
```

This example begins by creating two variables, X and Y, to hold the row

and column value of the table. X is the row variable and Y is the column variable.

To make the table readable, this example must create a heading at the top and another along the side. When users see a 1 at the top and a 1 at the side, and follow these values to where they intersect in the table, they can see the value of the two numbers when multiplied.

The first `print()` statement adds a space (because nothing appears in the corner of the table; see Figure 9-8 to more easily follow this discussion). All the formatting statement says is to create a space 4 characters wide and place a space within it. The `{:>4}` part of the code determines the size of the column. The `format(' ')` function determines what appears in that space. The `end` attribute of the `print()` statement changes the ending character from a carriage return to a simple space.

The first `for` loop displays the numbers 1 through 10 at the top of the table. The `range()` function creates the sequence of numbers for you. When using the `range()` function, you specify the starting value, which is 1 in this case, and one more than the ending value, which is 11 in this case.

At this point, the cursor is sitting at the end of the heading row. To move it to the next line, the code issues a `print()` call with no other information. Even though the next bit of code looks quite complex, you can figure it out if you look at it a line at a time. The multiplication table shows the values from $1 * 1$ to $10 * 10$, so you need ten rows and ten columns to display the information. The `for` statement tells Python to create ten rows.

Look again at Figure 9-8 to note the row heading. The first `print()` call displays the row heading value. Of course, you have to format this information, and the code uses a space of four characters that end with a space, rather than a carriage return, in order to continue printing information in that row.

The `while` loop comes next. This loop prints the columns in an individual row. The column values are the multiplied values of $X * Y$. Again, the

output is formatted to take up four spaces. The while loop ends when Y is updated to the next value by using $Y+=1$.

.t() statement ends the current
of the next row, which begins with 1.

2. Click Run Cell.

4 Working on Text Data

Text is one of the most widespread forms of sequence data. It can be understood as either a sequence of characters or a sequence of words, but it's most common to work at the level of words. The deep-learning sequence processing models introduced in the following sections can use text to produce a basic form of natural-language understanding, sufficient for applications including document classification, sentiment analysis, author identification, and even question-answering (QA) (in a constrained context). Of course, keep in mind throughout this chapter that none of these deep learning models truly understand text in a human sense; rather, these models can map the statistical structure of written language, which is sufficient to solve many simple textual tasks. Deep learning for natural-language processing is pattern recognition applied to words, sentences, and paragraphs, in much the same way that computer vision is pattern recognition applied to pixels.

Like all other neural networks, deep learning models don't take as input raw text: they only work with numeric tensors. Vectorizing text is the process of transforming text into numeric tensors. This can be done in multiple ways:

Segment text into words, and transform each word into a vector. Segment text into characters, and transform each character into a vector.

Extract n-grams of words or characters, and transform each n-gram into a vector. N-grams are overlapping groups of multiple consecutive words or characters.

Collectively, the different units into which you can break down text (words, characters, or n-grams) are called tokens, and breaking text into such tokens is called tokenization. All text-vectorization processes consist of applying some tokenization scheme and then associating numeric vectors with the generated tokens. These vectors, packed into sequence tensors, are fed into deep neural networks. There are multiple ways to associate a vector with a token. In this section, I'll present two major ones: one-hot encoding of tokens,

and

Text

token embedding (typically used exclusively

for "The cat sat on the mat."

words, and called word embedding). The remainder of this section explains these

Tokens

techniques and shows how to use them to "the", "cat", "sat", "on", "the", "mat", "."

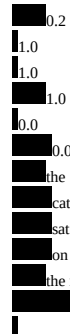
go from raw text to a Numpy tensor that you

can Vector encoding of the tokens

send to a Keras network.



Figure 6.1 From text to tokens to vectors_{1.0}



Understanding n-grams and bag-of-words

Word n-grams are groups of N (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence "The cat sat on the mat." It may be decomposed into the following set the mat." It may be decomposed into the following set grams:

{"The", "The cat", "cat", "cat sat", "sat",

"sat on", "on", "on the", "the", "the mat", "mat"]}

It may also be decomposed into the following set of 3-grams: {"The", "The cat", "cat", "cat sat", "The cat sat",

"sat", "sat on", "on", "cat sat on", "on the", "the",

"sat on the", "the mat", "mat", "on the mat"]}

Such a set is called a bag-of-2-grams or bag-of-3-grams, respectively. The term bag here refers to the fact that you're dealing with a set of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization methods is called bag-of-words.

Because bag-of-words isn't an order-preserving tokenization method (the tokens generated are understood as a set, not a sequence, and the general structure of the sentences is lost), it tends to be used in shallow language-processing models rather than in deep-learning models. Extracting n-grams is a form of feature engineering, and deep learning does away with this kind of rigid, brittle approach, replacing it with hierarchical feature learning. One-dimensional convnets and recurrent neural networks, introduced later in this chapter, are capable of learning representations for groups of words and characters without being explicitly told about the existence of such groups, by looking at continuous word or character sequences. For this reason, we won't cover n-grams any further in this book. But do keep in mind that they're a powerful, unavoidable feature-engineering tool when using lightweight, shallow text-processing models such as logistic regression and random forests.

One-hot encoding of words and characters

One-hot encoding is the most common, most basic way to turn a token into a vector. You saw it in action in the initial IMDB and Reuters examples in chapter 3 (done with words, in that case). It consists of associating a unique integer index with every word and then turning this integer index i into a binary vector of size N (the size of the vocabulary); the vector is all zeros except for the i th entry, which is 1.

Of course, one-hot encoding can be done at the character level, as well. To unambiguously drive home what one-hot encoding is and how to implement it, listings 6.1 and 6.2 show two toy examples: one for words, the other for

characters.

Listing 6.1 Word-level one-hot encoding (toy example)

Builds an index of all tokens in the data

Tokenizes the samples via the split method. In real life, you'd also strip punctuation and special characters but it could be an entire document from the samples.

```
import numpy as np
samples = ['The cat sat on the mat.',
           'The dog ate my homework.']
```

```
token_index = {}
for sample in samples:
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1
max_length = 10
results = np.zeros(shape=(len(samples),
                           max_length,
                           max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.
```

This is where you store the results.

Assigns a unique index to each

unique word. Note that you don't Vectorizes the samples. You'll only attribute index 0 to anything. consider the first max_length words in each sample.

Listing 6.2 Character-level one-hot encoding (toy example)

```
import string
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable
token_index = dict(zip(range(1, len(characters) + 1),
                       characters))
```

```
max_length = 50
results = np.zeros((len(samples), max_length,
                    max(token_index.keys()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

characters

Note that Keras has built-in utilities for doing one-hot encoding of text at the word level or character level, starting from raw text data. You should use these utilities, because they take care of a number of important features such as stripping special characters from strings and only taking into account the N most common words in your dataset (a common restriction, to avoid dealing with very large input vector spaces).

Listing 6.3 Using Keras for word-level one-hot encoding **Creates a tokenizer, configured to only take into account the**

```
from keras.preprocessing.text import Tokenizer 1,000 most common words
samples = ['The cat sat on the mat.', 'The dog ate my homework.'] tokenizer =
Tokenizer(num_words=1000) tokenizer.fit_on_texts(samples) Builds Turns
strings into lists these sequences =
tokenizer.texts_to_sequences(samples) of integer indices word
index one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')
word_index = tokenizer.word_index print('Found %s unique tokens.' %
len(word_index))
How you can recover
```

You could also directly get the one-hot ^{the word index that} **binary**
representations. Vectorization ^{was computed} **modes other than onehot**
encoding are supported by this tokenizer.

A variant of one-hot encoding is the so-called one-hot hashing trick, which you can use when the number of unique tokens in your vocabulary is too large to handle explicitly. Instead of explicitly assigning an index to each word and keeping a reference of these

indices in a dictionary, you can hash words into vectors of fixed size. This is typically done with a very lightweight hashing function. The main advantage of this method is that it does away with maintaining an explicit word index, which saves memory and allows online encoding of the data (you can generate token vectors right away, before you've seen all of the available data). The one drawback of this approach is that it's susceptible to hash collisions: two different words may end up with the same hash, and subsequently any machinelearning model looking at these hashes won't be able to tell the difference between these words. The likelihood of hash

collisions decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed.

Listing 6.4 Word-level one-hot encoding with hashing trick (toy example)

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']  
dimensionality = 1000 max_length = 10
```

```
results = np.zeros((len(samples), max_length, dimensionality))  
for i, sample in enumerate(samples):  
    for j, word in list(enumerate(sample.split()))[:max_length]:  
        index = abs(hash(word)) % dimensionality  
        results[i, j, index] = 1.
```

Stores the words as vectors of size 1,000. If you have close to 1,000 words (or more), you'll see many hash collisions, which will decrease the accuracy of this encoding

method.

Hashes the

word into a

random integer index between 0 and 1,000

Using word embeddings

Another popular and powerful way to associate a vector with a word is the use of dense word vectors, also called word embeddings. Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (same dimensionality as the number of words in the vocabulary), word embeddings are lowdimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors); see figure 6.2. Unlike the word vectors obtained via one-hot encoding, word embeddings are learned from data. It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

Sparse

High-dimensional Hardcoded
Dense

Lower
dimensional

Learned from data
sparse, high-dimensional, and hardcoded, word embeddings are dense,
relatively
lowdimensional, and learned from data.

Figure 6.2



obtained from one-hot Word embeddings: word vectors:
Whereas word representations encoding or hashing are One-hot

There are two ways to obtain word embeddings:

Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.

Load into your model word embeddings that were precomputed using a different machinelearning task than the one you're trying to solve. These are called pretrained word embeddings.

Let's look at both.

LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER

The simplest way to associate a dense vector with a word is to choose the vector at random. The problem with this approach is that the resulting embedding space has no structure: for instance, the words accurate and exact may end up with completely different embeddings, even though they're interchangeable in most sentences. It's difficult for a deep neural network to make sense of such a noisy, unstructured embedding space.

To get a bit more abstract, the geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space. For instance, in a reasonable embedding space, you would expect synonyms to be embedded into similar word vectors; and in general, you would expect the geometric distance (such as L2 distance) between any two word vectors to relate to the semantic distance between the associated words (words meaning different things are embedded at points far away from each other, whereas related words are closer). In addition to distance, you may want specific directions in the embedding space to be meaningful. To make this clearer, let's look at a concrete example.

In figure 6.3, four words are embedded on a 2D plane: cat, dog, wolf, and tiger. With the vector

1

Wolf

Tiger

representations we chose here, some semantic relationships between these words can be

Cat

same vector allows us to go from cat to tiger and from

x

0

dog to wolf: this vector could be interpreted as the “from pet to wild animal” vector. encoded as geometric transformations. For instance, the Similarly, another vector

lets us go from dog to cat and from wolf to tiger, which could be interpreted as a “from canine to feline” vector.

In real-world word-embedding spaces, common exam
0 1

Figure 6.3 A toy example of a word-embedding space

ples of meaningful geometric transformations are “gender” vectors and “plural” vectors. For instance, by adding a “female” vector to the vector “king,” we obtain the vector “queen.” By adding a “plural” vector, we obtain “kings.” Word- embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

Is there some ideal word-embedding space that would perfectly map human language and could be used for any natural-language-processing task? Possibly, but we have yet to compute anything of the sort. Also, there is no such a thing as human language—there are many different languages, and they aren't isomorphic, because a language is the reflection of a specific culture and a specific context. But more pragmatically, what makes a good word-embedding space depends heavily on your task: the perfect word-embedding space for an English-language movie-review sentiment-analysis model may look different from the perfect embedding space for an English-language legal-document-classification model, because the importance of certain semantic relationships varies from task to task.

It's thus reasonable to learn a new embedding space with every new task. Fortunately, backpropagation makes this easy, and Keras makes it even easier. It's about learning the weights of a layer: the Embedding layer.

Listing 6.5 Instantiating an **Embedding** layer

```
from keras.layers import Embedding
```

The Embedding layer takes at least two

arguments: the number of possible tokens

```
embedding_layer = Embedding(1000, 64)
```

(here, 1,000: 1 + maximum word index)

and the dimensionality of the embeddings (here, 64).

The Embedding layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup (see figure 6.4).

Word index Embedding layer Corresponding word vector

Figure 6.4 The **Embedding** layer

The Embedding layer takes as input a 2D tensor of integers, of shape (samples, sequence_length), where each entry is a sequence of integers. It can embed sequences of variable lengths: for instance, you could feed into the Embedding layer in the previous example batches with shapes (32, 10) (batch of 32 sequences of length 10) or (64, 15) (batch of 64 sequences of length 15). All sequences in a batch must have the same length, though (because you

need to pack them into a single tensor), so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

This layer returns a 3D floating-point tensor of shape (samples, sequence_length, embedding_dimensionality). Such a 3D tensor can then be processed by an RNN layer or a 1D convolution layer (both will be introduced in the following sections).

When you instantiate an Embedding layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer. During training, these word vectors are gradually adjusted via backpropagation, structuring the space into something the downstream model can exploit. Once fully trained, the embedding space will show a lot of structure—a kind of structure specialized for the specific problem for which you’re training your model.

Let’s apply this idea to the IMDB movie -review sentiment-prediction task that you’re already familiar with. First, you’ll quickly prepare the data. You’ll restrict the movie reviews to the top 10,000 most common words (as you did the first time you worked with this dataset) and cut off the reviews after only 20 words. The network will learn 8-dimensional embeddings for each of the 10,000 words, turn the input integer sequences (2D integer tensor) into embedded sequences (3D float tensor), flatten the tensor to 2D, and train a single Dense layer on top for classification.

Listing 6.6 Loading the IMDB data for use with an **Embedding** layer from `keras.datasets` import `imdb`

Cuts off the text after this number of words (among

from `keras` import

```
from keras.preprocessing import sequence
max_features = 10000
maxlen = 20
(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features)
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
```

Loads the data as lists of integers
Turns the

lists of integers into a 2D integer tensor of shape (samples, maxlen)

Listing 6.7 Using an **Embedding** layer and classifier on the IMDB data

Specifies the maximum input length to the Embedding layer so you can later flatten the embedded inputs. After the Embedding layer, the activations have shape

8). Flattens the 3D tensor of embeddings into a 2D

from keras.models import

tensor of shape (samples,

maxlen * 8)

from keras.layers import Flatten, Dense

model = Sequential()

model.add(Embedding(10000, 8, input_length=maxlen))

(samples, maxlen,

Sequential

model.add(Flatten()) **Adds the**

model.add(Dense(1, activation='sigmoid')) **classifier on top**

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc']) model.summary()

history = model.fit(x_train, y_train,

epochs=10, batch_size=32, validation_split=0.2)

You get to a validation accuracy of ~76%, which is pretty good considering that you're only looking at the first 20 words in every review. But note that merely flattening the embedded sequences and training a single Dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and sentence structure (for example, this model would likely treat both "this movie is a bomb" and "this movie is the bomb" as being negative reviews). It's much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole.

That's what we'll focus on in the next few sections.

USING PRETRAINED WORD EMBEDDINGS

Sometimes, you have so little training data available that you can't use your data alone to learn an appropriate task-specific embedding of your vocabulary. What do you do then?

Instead of learning word embeddings jointly with the problem you want to solve, you can load embedding vectors from a precomputed embedding space that you know is highly structured and exhibits useful properties—that captures generic aspects of language structure. The rationale behind using pretrained word embeddings in natural-language processing is much the same as for using pretrained convnets in image classification: you don't have enough data available to learn truly powerful features on your own, but you expect the features that you need to be fairly generic—that is, common visual features or semantic features. In this case, it makes sense to reuse features learned on a different problem.

Such word embeddings are generally computed using word-occurrence statistics (observations about what words co-occur in sentences or documents), using a variety of techniques, some involving neural networks, others not. The idea of a dense, lowdimensional embedding space for words, computed in an unsupervised way, was initially explored by Bengio et al. in the early 2000s,⁶ but it only started to take off in research and industry applications after the release of one of the most famous and successful wordembedding schemes: the Word2vec algorithm (<https://code.google.com/archive/p/word2vec>), developed by Tomas Mikolov at Google in 2013. Word2vec dimensions capture specific semantic properties, such as gender.

There are various precomputed databases of word embeddings that you can download and use in a Keras Embedding layer. Word2vec is one of them. Another popular one is called Global Vectors for Word Representation (GloVe, <https://nlp.stanford.edu/projects/glove>), which was developed by Stanford researchers in 2014. This embedding technique is based on factorizing a matrix of word co-occurrence statistics. Its developers have made available precomputed embeddings for millions of English tokens, obtained from Wikipedia data and Common Crawl data.

⁶ Yoshua Bengio et al., *Neural Probabilistic Language Models* (Springer, 2003).

Let's look at how you can get started using GloVe embeddings in a Keras model. The same method is valid for Word2vec embeddings or any other wordembedding database. You'll also use this example to refresh the texttokenization techniques introduced a few paragraphs ago: you'll start from raw text and work your way up.

Putting it all together: from raw text to word embeddings

You'll use a model similar to the one we just went over: embedding sentences in sequences of vectors, flattening them, and training a Dense layer on top. But you'll do so using pretrained word embeddings; and instead of using the pretokenized IMDB data packaged in Keras, you'll start from scratch by downloading the original text data.

DOWNLOADING THE IMDB DATA AS RAW TEXT

First, head to <http://mng.bz/0tIo> and download the raw IMDB dataset. Uncompress it.

Now, let's collect the individual training reviews into a list of strings, one string per review. You'll also collect the review labels (positive/negative) into a labels list.

Listing 6.8 Processing the labels of the raw IMDB data

```
import os

imdb_dir = '/Users/fchollet/Downloads/aclImdb' train_dir =
os.path.join(imdb_dir, 'train')

labels = [] texts = []
for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
```

else:

labels.append(1)

TOKENIZING THE DATA

Let's vectorize the text and prepare a training and validation split, using the concepts introduced earlier in this section. Because pretrained word embeddings are meant to be particularly useful on problems where little training data is available (otherwise, task-specific embeddings are likely to outperform them), we'll add the following twist: restricting the training data to the first 200 samples. So you'll learn to classify movie reviews after looking at just 200 examples.

Listing 6.9 Tokenizing the text of the raw IMDB data

from keras.preprocessing.text import Tokenizer from

keras.preprocessing.sequence import pad_sequences import numpy as np

```
maxlen = 100 Cuts off reviews after 100 words training_samples = 200 Trains  
on 200 samples validation_samples = 10000 Validates on 10,000 samples  
max_words = 10000
```

Considers only the top

tokenizer = Tokenizer(num_words=max_words)

10,000 words in the dataset

tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index

print('Found %s unique tokens.' % len(word_index)) data =

pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels) print('Shape of data tensor:', data.shape)

print('Shape of label tensor:', labels.shape) indices = np.arange(data.shape[0])

Splits the data into a training

set, but first shuffles the

you're starting with data in which

set and a

np.random.shuffle(indices) **validation data**, data = data[indices] **because**

```
labels = labels[indices] samples are ordered (all negative first, then all positive)
```

```
x_train = data[:training_samples] y_train = labels[:training_samples]
```

```
x_val = data[training_samples: training_samples + validation_samples] y_val = labels[training_samples: training_samples + validation_samples]
```

DOWNLOADING THE GLOVE WORD EMBEDDINGS

Go to <https://nlp.stanford.edu/projects/glove>, and download the precomputed embeddings from 2014 English Wikipedia. It's an 822 MB zip file called glove.6B.zip, containing 100-dimensional embedding vectors for 400,000 words (or nonword tokens). Unzip it.

PREPROCESSING THE EMBEDDINGS

Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation (as number vectors).

Listing 6.10 Parsing the GloVe word-embeddings file

```
glove_dir = '/Users/fchollet/Downloads/glove.6B'
```

```
embeddings_index = {} f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))  
for line in f:
```

```
    values = line.split() word = values[0] coefs = np.asarray(values[1:],  
dtype='float32') embeddings_index[word] = coefs  
f.close() print('Found %s word vectors.' % len(embeddings_index))
```

Next, you'll build an embedding matrix that you can load into an Embedding layer. It must be a matrix of shape (max_words, embedding_dim), where each entry *i* contains the embedding_dim-dimensional vector for the word of index *i* in the reference word index (built during tokenization). Note that index 0 isn't supposed to stand for any word or token—it's a placeholder.

Listing 6.11 Preparing the GloVe word-embeddings matrix

```
embedding_dim = 100
```

```
embedding_matrix = np.zeros((max_words, embedding_dim))  
for word, i in word_index.items():  
    if i < max_words:
```

```
embedding_vector =  
embeddings_index.get(word)Words not found in the if embedding_vector is  
not None:embedding index will embedding_matrix[i] =  
embedding_vectorbe all zeros.
```

DEFINING A MODEL

You'll use the same model architecture as before.

Listing 6.12 Model definition

```
from keras.models import Sequential from keras.layers import Embedding,  
Flatten, Dense
```

```
model = Sequential() model.add(Embedding(max_words, embedding_dim,  
input_length=maxlen)) model.add(Flatten()) model.add(Dense(32,  
activation='relu')) model.add(Dense(1, activation='sigmoid'))  
model.summary()
```

LOADING THE GLOVE EMBEDDINGS IN THE MODEL

The Embedding layer has a single weight matrix: a 2D float matrix where each entry i is the word vector meant to be associated with index i . Simple enough. Load the GloVe matrix you prepared into the Embedding layer, the first layer in the model.

Listing 6.13 Loading pretrained word embeddings into the

Embedding layer

```
model.layers[0].set_weights([embedding_matrix]) model.layers[0].trainable  
= False
```

Additionally, you'll freeze the Embedding layer (set its trainable attribute to False), following the same rationale you're already familiar with in the context of pretrained convnet features: when parts of a model are pretrained (like your Embedding layer) and parts are randomly initialized (like your classifier), the pretrained parts shouldn't be updated during training, to avoid forgetting what they already know. The large gradient updates triggered by the randomly initialized layers would be disruptive to the already-learned features.

TRAINING AND EVALUATING THE MODEL

Compile and train the

model.

Listing 6.14 Training and evaluation

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[  
    'acc'])
```

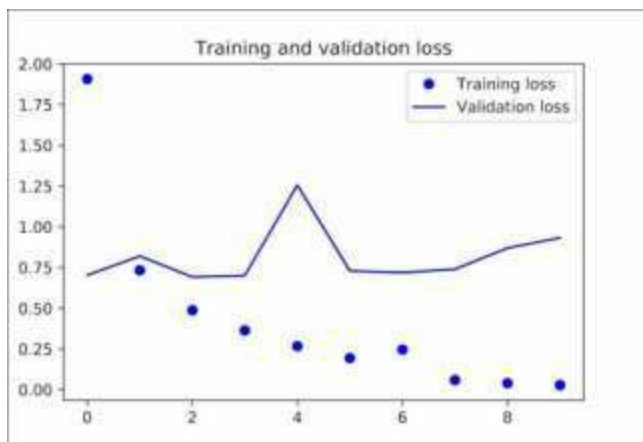
```
history = model.fit(x_train, y_train,  
    epochs=10, batch_size=32, validation_data=(x_val, y_val))  
model.save_weights('pre_trained_glove_model.h5')
```

Now, plot the model's performance over time (see figures 6.5 and 6.6).

Listing 6.15 Plotting the results

```
import matplotlib.pyplot as plt
```

```
acc = history.history['acc'] val_acc = history.history['val_acc'] loss =  
history.history['loss'] val_loss = history.history['val_loss'] epochs = range(1,  
len(acc) + 1)
```

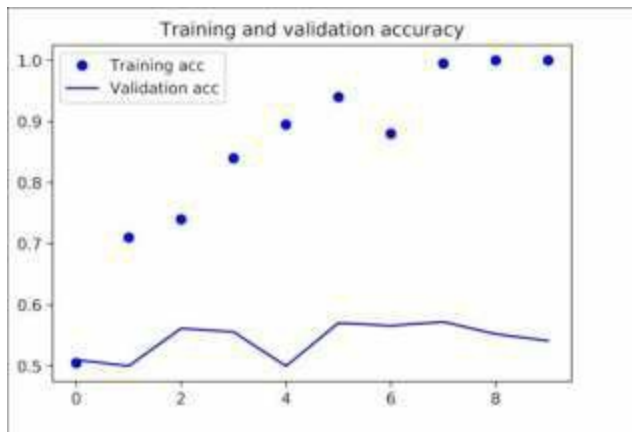


```
plt.plot(epochs, acc, 'bo', label='Training acc')  
plt.plot(epochs, val_acc, 'b', label='Validation acc')  
plt.title('Training and validation accuracy') plt.legend()  
plt.figure()
```

```
plt.plot(epochs, loss, 'bo', label='Training loss')  
plt.plot(epochs, val_loss, 'b', label='Validation loss')
```

```
plt.title('Training and validation loss') plt.legend() plt.show()
```

Figure 6.5 Training and validation loss when using pretrained word



embeddings Figure 6.6 Training and validation accuracy when using pretrained word embeddings

The model quickly starts overfitting, which is unsurprising given the small number of training samples. Validation accuracy has high variance for the same reason, but it seems to reach the high 50s.

Note that your mileage may vary: because you have so few training samples, performance is heavily dependent on exactly which 200 samples you choose—and you’re choosing them at random. If this works poorly for you, try choosing a different random set of 200 samples, for the sake of the exercise (in real life, you don’t get to choose your training data).

You can also train the same model without loading the pretrained word embeddings and without freezing the embedding layer. In that case, you’ll learn a task-specific embedding of the input tokens, which is generally more powerful than pretrained word embeddings when lots of data is available. But in this case, you have only 200 training samples. Let’s try it (see figures 6.7 and 6.8).

Listing 6.16 Training the same model without pretrained word embeddings



from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

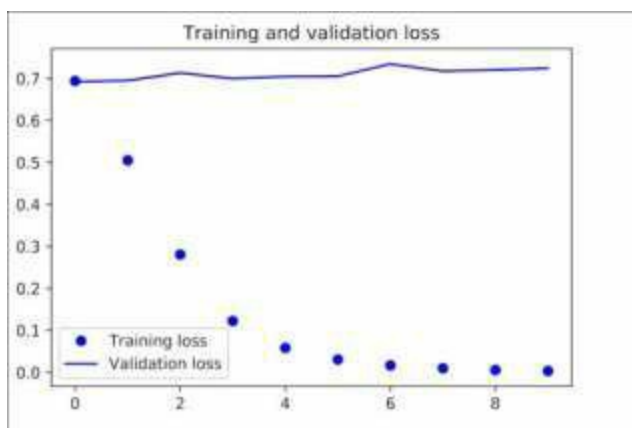
```
model = Sequential()
```

```
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))  
model.add(Flatten())  
model.add(Dense(32, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))  
model.summary()
```

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

```
history = model.fit(x_train, y_train,
```

```
epochs=10, batch_size=32, validation_data=(x_val, y_val))
```



Training and validation loss without using pretrained word

embeddings

Figure 6.8 Training and validation accuracy without using pretrained word embeddings

Validation accuracy stalls in the low 50s. So in this case, pretrained word embeddings outperform jointly learned embeddings. If you increase the number of training samples, this will quickly stop being the case—try it as an exercise.

Finally, let's evaluate the model on the test data. First, you need to tokenize the test data.

Listing 6.17 Tokenizing the data of the test set

```
test_dir = os.path.join(imdb_dir, 'test')
labels = [] texts = []
for label_type in ['neg', 'pos']:

    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
Next, load and evaluate the first model.
```

Listing 6.18 Evaluating the model on the test set

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

You get an appalling test accuracy of 56%. Working with just a handful of training samples is difficult!

Wrapping up

Now you're able to do the following:

Turn raw text into something a neural network can process

Use the Embedding layer in a Keras model to learn task-specific token embeddings

Use pretrained word embeddings to get an extra boost on small natural language-processing problems

Understanding recurrent neural networks

A major characteristic of all neural networks you've seen so far, such as densely connected networks and convnets, is that they have no memory. Each input shown to them is processed independently, with no state kept in between inputs. With such networks, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: turn it into a single data point. For instance, this is what you did in the IMDB example: an entire movie review was transformed into a single large vector and processed in one go. Such networks are called feedforward networks.

In contrast, as you're reading the present sentence, you're processing it word by word—or rather, eye saccade by eye saccade—while keeping memories of what came before; this gives you a fluid representation of the meaning conveyed by this sentence. Biological intelligence processes information incrementally while maintaining an internal model of what it's processing, built from past information and constantly updated as new information comes in.

A recurrent neural network (RNN) adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a state containing^{Output}

■■■■

information relative to what it has seen so far. In effect, an RNN is a type of neural network that has an_{RNN}

■■■

internal loop (see figure 6.9). The state of the RNN is ^{Recurrent}

connection

reset between processing two different, independent

sequences (such as two different IMDB reviews), so^{Input} you still consider one sequence a single data point: a single input to the network. What changes is that this data point is no longer processed in a Figure 6.9 A recurrent single step; rather, the network internally loops over network: a network with a loop sequence elements.

To make these notions of loop and state clear, let's implement the forward pass of a toy RNN in Numpy. This RNN takes as input a sequence of vectors, which you'll encode as a 2D tensor of size (timesteps, input_features). It loops over timesteps, and at each timestep, it considers its current state at t and the input at t (of shape (input_features,)), and combines them to obtain the output at t . You'll then set the state for the next step to be this previous output. For the first timestep, the previous output isn't defined; hence, there is no current state. So, you'll initialize the state as an allzero vector called the initial state of the network. In pseudocode, this is the RNN.

Listing 6.19 Pseudocode RNN

```
state_t = 0 The state at t
for input_t in input_sequence: Iterates over sequence elements output_t =
f(input_t, state_t) state_t = output_t The previous output becomes the state
for the next iteration.
```

You can even flesh out the function f : the transformation of the input and state into an output will be parameterized by two matrices, W and U , and a bias vector. It's similar to the transformation operated by a densely connected layer in a feedforward network.

Listing 6.20 More detailed pseudocode for the RNN

```
state_t = 0 for input_t in input_sequence:
output_t = activation(dot(W, input_t) + dot(U, state_t) + b) state_t = output_t
```

To make these notions absolutely unambiguous, let's write a naive Numpy implementation of the forward pass of the simple RNN.

Listing 6.21 Numpy implementation of a simple RNN **Number of timesteps in Dimensionality of the the input sequence** import numpy as

input feature space

Input `timesteps = 100` **Dimensionality of the noise for**
`input_features = 32` **output feature space**

`output_features = 64`

Initial state: an all-zero vector

`inputs = np.random.random((timesteps, input_features))`

`state_t = np.zeros((output_features,))` `W =`
`np.random.random((output_features, input_features))`

Creates random weight matrices `input_t` **is a vector of data: random**

the sake of the example

`U =`
`np.random.random((output_features, output_features))` `b =`
`np.random.random((output_features,))`
`successive_outputs = []` **shape (input_features,).**

`for`
`input_t in inputs:` `output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) +`
`b)` `successive_outputs.append(output_t)` `state_t = output_t`

Stores this output in a list The final output is a 2D tensor of

`final_output_sequence=np.concatenate(successive_outputs,axis=0)`

shape (timesteps, output_features).

Combines the input with the current

state (the previous output) to obtain **Updates the state of the**

Easy enough: in summary, an RNN is a for loop that reuses quantities computed during the previous iteration of the loop, nothing more. Of course, there are many different RNNs fitting this definition that you could build—this example is one of the simplest RNN formulations. RNNs are

characterized by their step function, such as the following function in this case (see figure 6.10):

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

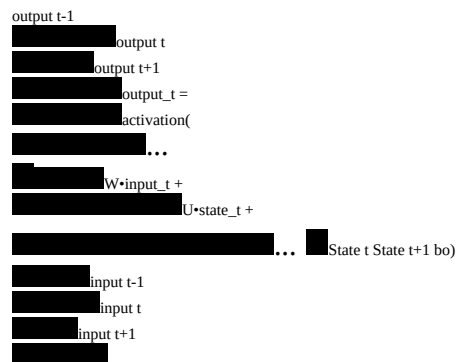


Figure 6.10 A simple RNN, unrolled over time

NOTE In this example, the final output is a 2D tensor of shape (timesteps, output_features), where each timestep is the output of the loop at time t . Each timestep t in the output tensor contains information about timesteps 0 to t in the input sequence—about the entire past. For this reason, in many cases, you don’t need this full sequence of outputs; you just need the last output ($output\_t$ at the end of the loop), because it already contains information about the entire sequence.

A recurrent layer in Keras

The process you just naively implemented in Numpy corresponds to an actual Keras layer—the SimpleRNN layer: `from keras.layers import SimpleRNN`

There is one minor difference: SimpleRNN processes batches of sequences, like all other

Keras layers, not a single sequence as in the Numpy example. This means it takes inputs of shape (batch_size, timesteps, input_features), rather than (timesteps, input_features).

Like all recurrent layers in Keras, SimpleRNN can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)) or only the last output for each input sequence (a 2D tensor of shape

(batch_size, output_features)). These two modes are controlled by the return_sequences constructor argument. Let's look at an example that uses SimpleRNN and returns only the output at the last timestep:

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))

>>> model.add(SimpleRNN(32)) >>> model.summary()
```

Layer (type) Output Shape Param #

=====
embedding_22 (Embedding) (None, None, 32) 320000

simplernn_10 (SimpleRNN) (None, 32) 2080

=====
Total params: 322,080

Trainable params: 322,080

Non-trainable params: 0

The following example returns the full state sequence:

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
```

Layer (type) Output Shape Param #

embedding_23 (Embedding) (None, None, 32) 320000

—
simplernn_11 (SimpleRNN) (None, None, 32) 2080

=====
=====

Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0

It's sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get all of the intermediate layers to return full sequence of outputs: >>> model = Sequential()

```
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
```

```
>>> model.add(SimpleRNN(32,
return_sequences=True))
```

Last layer only returns

```
>>> model.add(SimpleRNN(32))
```

the last output

```
>>> model.summary()
```

—
Layer (type) Output Shape Param #

=====
=====
embedding_24 (Embedding) (None, None, 32) 320000

—
simplernn_12 (SimpleRNN) (None, None, 32) 2080

— simplernn_13 (SimpleRNN) (None, None, 32) 2080

__ simplernn_14 (SimpleRNN) (None, None, 32) 2080

__
simplernn_15 (SimpleRNN) (None, 32) 2080

=====

Total params: 328,320

Trainable params: 328,320 Non-trainable params: 0

Now, let's use such a model on the IMDB movie-review-classification problem. First, preprocess the data.

Listing 6.22 Preparing the IMDB data

```
from keras.datasets import imdb from keras.preprocessing import sequence
```

```
Number of words to
```

```
max_features = 10000consider as features
```

```
maxlen = 500
```

```
Cuts off texts after this many words (among
```

```
batch_size = 32
```

```
the max_features most common words)
```

```
print('Loading data...')
```

```
(input_train, y_train), (input_test, y_test) = imdb.load_data(
```

```
num_words=max_features)
```

```
print(len(input_train), 'train sequences') print(len(input_test), 'test sequences')
```

```
print('Pad sequences (samples x time)') input_train =
```

```
sequence.pad_sequences(input_train, maxlen=maxlen) input_test =
```

```
sequence.pad_sequences(input_test, maxlen=maxlen) print('input_train
```

```
shape:', input_train.shape) print('input_test shape:', input_test.shape)
```

Let's train a simple recurrent network using an Embedding layer and a SimpleRNN layer.

Listing 6.23 Training the model with **Embedding** and **SimpleRNN** layers

```
from keras.layers import Dense
```

```
model = Sequential() model.add(Embedding(max_features, 32))
```

```
model.add(SimpleRNN(32)) model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=
['acc']) history = model.fit(input_train, y_train,
```

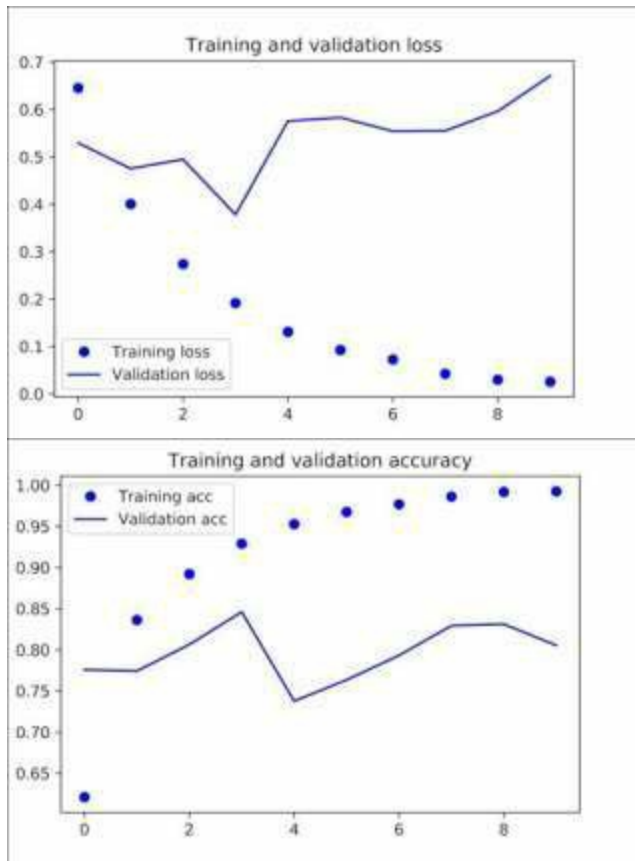
```
epochs=10, batch_size=128, validation_split=0.2)
```

Now, let's display the training and validation loss and accuracy (see figures 6.11 and 6.12).

Listing 6.24 Plotting results

```
import matplotlib.pyplot as plt
```

```
acc = history.history['acc'] val_acc = history.history['val_acc'] loss =
```



with **SimpleRNN**

```
history.history['loss'] val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
```

```
plt.plot(epochs, acc, 'bo', label='Training acc')
```

```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
```

```
plt.title('Training and validation accuracy') plt.legend()  
plt.figure()
```

```
plt.plot(epochs, loss, 'bo', label='Training loss')  
plt.plot(epochs, val_loss, 'b', label='Validation loss')  
plt.title('Training and validation loss') plt.legend() plt.show()
```

Figure 6.11 Training and validation loss on IMDB with **SimpleRNN**

Figure 6.12 Training and validation accuracy on IMDB As a reminder, in chapter 3, the first naive approach to this dataset got you to a test accuracy of 88%. Unfortunately, this small recurrent network doesn't perform well compared to this baseline (only 85% validation accuracy). Part of the problem is that your inputs only consider the first 500 words, rather than full sequences—hence, the RNN has access to less information than the earlier baseline model. The remainder of the problem is that SimpleRNN isn't good at processing long sequences, such as text.

Other types of recurrent layers perform much better. Let's look at some more advanced layers.

Understanding the LSTM and GRU layers

SimpleRNN isn't the only recurrent layer available in Keras. There are two others: LSTM and GRU. In practice, you'll always use one of these, because SimpleRNN is generally too simplistic to be of real use. SimpleRNN has a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the vanishing gradient problem, an effect that is similar to what is observed with nonrecurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.⁷ The LSTM and GRU layers are designed to solve this problem.

Let's consider the LSTM layer. The underlying Long Short-Term Memory (LSTM) algorithm was developed by Hochreiter and Schmidhuber in 1997;⁸ it was the culmination of their research on the vanishing gradient problem.

This layer is a variant of the SimpleRNN layer you already know about; it adds a way to carry information across many timesteps. Imagine a conveyor belt running parallel to the sequence you're processing. Information from the sequence can jump onto the conveyor belt at any point, be transported to a later timestep, and jump off, intact, when you need it. This is essentially what LSTM does: it saves information for later, thus preventing older signals from gradually vanishing during processing.

⁷ See, for example, Yoshua Bengio, Patrice Simard, and Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent Is Difficult," *IEEE Transactions on Neural Networks* 5, no. 2 (1994).

⁸ Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9, no. 8 (1997).

To understand this in detail, let's start from the SimpleRNN cell (see figure 6.13). Because you'll have a lot of weight matrices, index the W and U matrices in the cell with the letter o (W_o and U_o) for output.

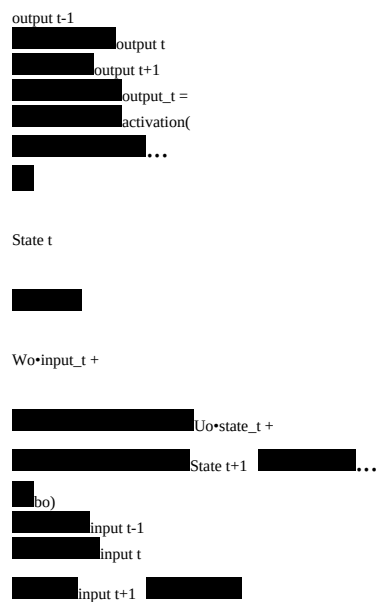


Figure 6.13 The starting point of an **LSTM** layer: a **SimpleRNN**

Let's add to this picture an additional data flow that carries information across timesteps. Call its values at different timesteps C_t , where C stands for carry. This information will have the following impact on the cell: it will be combined with the input connection and the recurrent connection (via a dense transformation: a dot product with a weight matrix followed by a bias add

and the application of an activation function), and it will affect the state being sent to the next timestep (via an activation function and a multiplication operation). Conceptually, the carry dataflow is a way to modulate the next output and the next state (see figure 6.14). Simple so far.

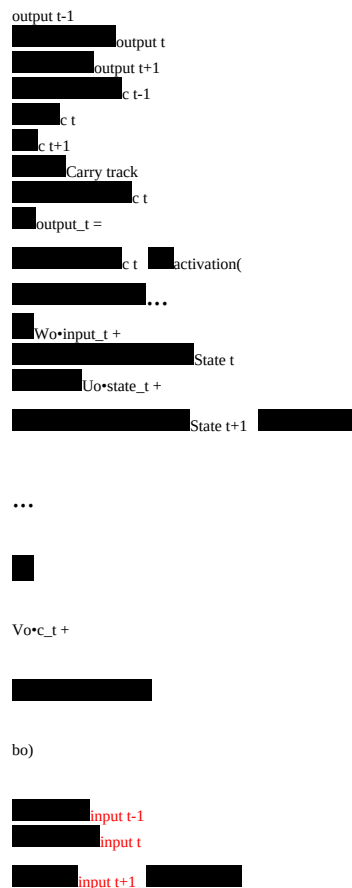


Figure 6.14 Going from a **SimpleRNN** to an **LSTM**: adding a carry track

Now the subtlety: the way the next value of the carry dataflow is computed. It involves three distinct transformations. All three have the form of a SimpleRNN cell:

$$y = \text{activation}(\text{dot}(\text{state}_t, U) + \text{dot}(\text{input}_t, W) + b)$$

But all three transformations have their own weight matrices, which you'll index with the letters i , f , and k . Here's what you have so far (it may seem a bit arbitrary, but bear with me).

Listing 6.25 Pseudocode details of the LSTM architecture (1/2) $\text{output}_t =$

$\text{activation}(\text{dot}(\text{state\_t}, U_o) + \text{dot}(\text{input\_t}, W_o) + \text{dot}(C_t, V_o) + b_o)$

$i_t = \text{activation}(\text{dot}(\text{state\_t}, U_i) + \text{dot}(\text{input\_t}, W_i) + b_i)$ $f_t =$

$\text{activation}(\text{dot}(\text{state\_t}, U_f) + \text{dot}(\text{input\_t}, W_f) + b_f)$ $k_t =$

$\text{activation}(\text{dot}(\text{state\_t}, U_k) + \text{dot}(\text{input\_t}, W_k) + b_k)$

You obtain the new carry state (the next c_t) by combining i_t , f_t , and k_t .

Listing 6.26 Pseudocode details of the LSTM architecture (2/2) $c_{t+1} = i_t * k_t + c_t * f_t$

Add this as shown in figure 6.15. And that's it. Not so complicated—merely a tad complex.

```

output t-1
    c t-1
    Compute c t new
    carry
    c t
    ...
    State t output t
    output t+1
    c t+1
    Carry track

```

Compute

```

    
```

new

```

    
```

carry

```

    output_t =
    c t
    activation(
    
```

$W_o \cdot \text{input\_t} +$

```

    
```

...

```

    
```

$U_o \cdot \text{state\_t} +$

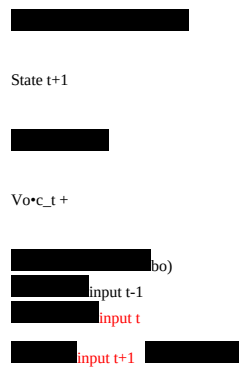


Figure 6.15 Anatomy of an **LSTM**

If you want to get philosophical, you can interpret what each of these operations is meant to do. For instance, you can say that multiplying c_t and f_t is a way to deliberately forget irrelevant information in the carry dataflow. Meanwhile, i_t and k_t provide information about the present, updating the carry track with new information. But at the end of the day, these interpretations don't mean much, because what these operations actually do is determined by the contents of the weights parameterizing them; and the weights are learned in an end-to-end fashion, starting over with each training round, making it impossible to credit this or that operation with a specific purpose. The specification of an RNN cell (as just described) determines your hypothesis space—the space in which you'll search for a good model configuration during training—but it doesn't determine what the cell does; that is up to the cell weights. The same cell with different weights can be doing very different things. So the combination of operations making up an RNN cell is better interpreted as a set of constraints on your search, not as a design in an engineering sense.

To a researcher, it seems that the choice of such constraints —the question of how to implement RNN cells—is better left to optimization algorithms (like genetic algorithms or reinforcement learning processes) than to human engineers. And in the future, that's how we'll build networks. In summary: you don't need to understand anything about the specific architecture of an LSTM cell; as a human, it shouldn't be your job to understand it. Just keep in mind what the LSTM cell is meant to do: allow past information to be reinjected at a later time, thus fighting the vanishing-gradient problem.

A concrete LSTM example in Keras

Now let's switch to more practical concerns: you'll set up a model using an LSTM layer and train it on the IMDB data (see figures 6.16 and 6.17). The network is similar to the one with SimpleRNN that was just presented. You only specify the output dimensionality of the LSTM layer; leave every other argument (there are many) at the Keras defaults. Keras has good defaults, and things will almost always “just work” without you having to spend time tuning parameters by hand.

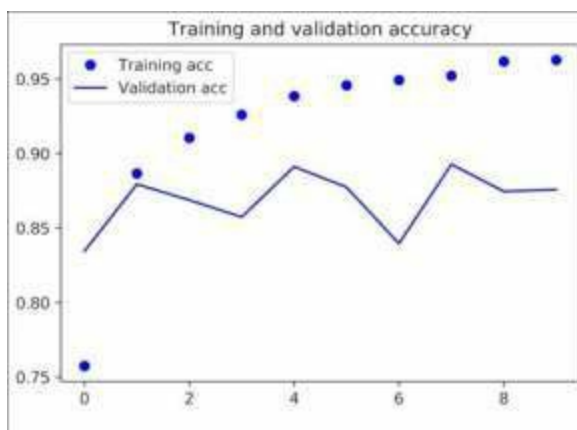
Listing 6.27 Using the **LSTM** layer in Keras

```
from keras.layers import LSTM
```



```
model = Sequential()
```

```
model.add(Embedding(max_features, 32)) model.add(LSTM(32))
```



```
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=
```


['acc'])

```
history = model.fit(input_train, y_train,  
epochs=10, batch_size=128, validation_split=0.2)
```

Figure 6.16 Training and validation loss on IMDB with LSTM

Figure 6.17 Training and validation accuracy on IMDB with LSTM

This time, you achieve up to 89% validation accuracy. Not bad: certainly much better than the SimpleRNN network—that’s largely because LSTM suffers much less from the vanishing-gradient problem—and slightly better than the fully connected approach from chapter 3, even though you’re looking at less data than you were in chapter 3. You’re truncating sequences after 500 timesteps, whereas in chapter 3, you were considering full sequences.

But this result isn’t groundbreaking for such a computationally intensive approach. Why isn’t LSTM performing better? One reason is that you made no effort to tune hyperparameters such as the embeddings dimensionality or the LSTM output dimensionality. Another may be lack of regularization. But honestly, the primary reason is that analyzing the global, long-term structure of the reviews (what LSTM is good at) isn’t helpful for a sentiment-analysis problem. Such a basic problem is well solved by looking at what words occur in each review, and at what frequency. That’s what the first fully connected approach looked at. But there are far more difficult natural language processing problems out there, where the strength of LSTM will become apparent: in particular, question-answering and machine translation.

Wrapping up

Now you understand the following:

What RNNs are and how they work

What LSTM is, and why it works better on long sequences than a naive RNN

How to use Keras RNN layers to process sequence data

Next, we’ll review a number of more advanced features of RNNs, which can help you get the most out of your deep-learning sequence models. Advanced use of recurrent neural networks

In this section, we'll review three advanced techniques for improving the performance and generalization power of recurrent neural networks. By the end of the section, you'll know most of what there is to know about using recurrent networks with Keras. We'll demonstrate all three concepts on a temperature-forecasting problem, where you have access to a timeseries of data points coming from sensors installed on the roof of a building, such as temperature, air pressure, and humidity, which you use to predict what the temperature will be 24 hours after the last data point. This is a fairly challenging problem that exemplifies many common difficulties encountered when working with timeseries.

We'll cover the following techniques:

Recurrent dropout—This is a specific, built-in way to use dropout to fight overfitting in recurrent layers.

Stacking recurrent layers—This increases the representational power of the network (at the cost of higher computational loads).

Bidirectional recurrent layers —These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

A temperature-forecasting problem

Until now, the only sequence data we've covered has been text data, such as the IMDB dataset and the Reuters dataset. But sequence data is found in many more problems than just language processing. In all the examples in this section, you'll play with a weather timeseries dataset recorded at the Weather Station at the Max Planck Institute for Biogeochemistry in Jena, Germany.⁹

In this dataset, 14 different quantities (such as air temperature, atmospheric pressure, humidity, wind direction, and so on) were recorded every 10 minutes, over several years. The original data goes back to 2003, but this example is limited to data from 2009–2016. This dataset is perfect for learning to work with numerical timeseries. You'll use it to build a model that takes as input some data from the recent past (a few days' worth of data points) and predicts the air temperature 24 hours in the future.

Download and uncompress the data as follows:

```
cd ~/Downloads mkdir jena_climate cd jena_climate wget
https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip Let's look at the data.
```

Listing 6.28 Inspecting the data of the Jena weather dataset

```
import os
```

```
data_dir = '/users/fchollet/Downloads/jena_climate' fname =
os.path.join(data_dir, 'jena_climate_2009_2016.csv')
```

```
f = open(fname) data = f.read() f.close()
lines = data.split('\n') header = lines[0].split(',') lines = lines[1:]
print(header) print(len(lines))
```

This outputs a count of 420,551 lines of data (each line is a timestep: a record of a date and 14 weather-related values), as well as the following header:

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

Now, convert all 420,551 lines of data into a Numpy array.

```
import numpy as np
```

```
float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]]
    float_data[i, :] = values
```

For instance, here is the plot of temperature (in degrees Celsius) over time (see figure 6.18). On this plot, you can clearly see the yearly periodicity of temperature.

Plotting the temperature timeseries
from matplotlib import pyplot as plt

```
temp = float_data[:, 1] <1> temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
```

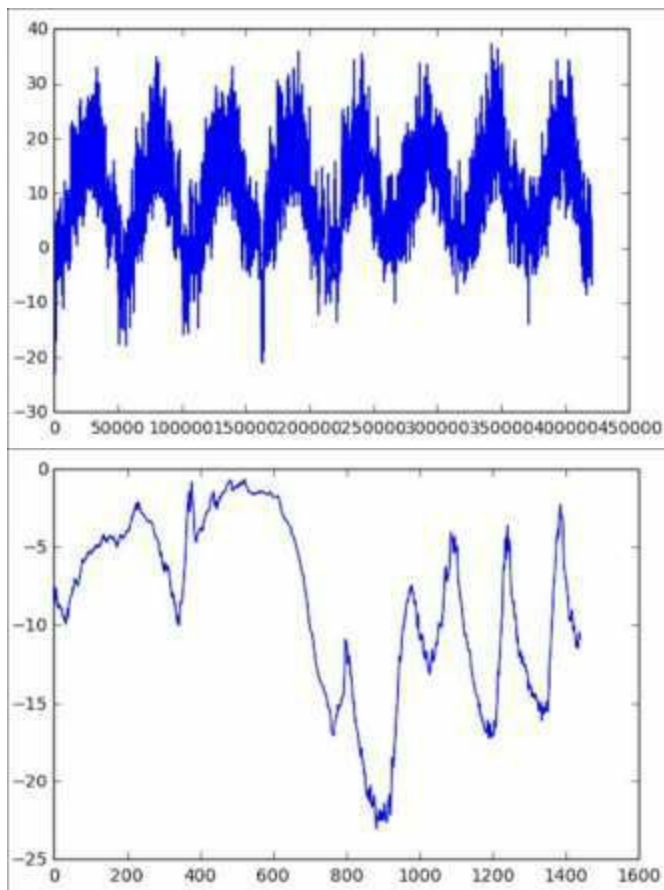


Figure 6.18
Temperature

over the full temporal range of the dataset (°C)

Here is a more narrow plot of the first 10 days of temperature data (see figure 6.19). Because the data is recorded every 10 minutes, you get 144 data points per day.

Listing 6.31 Plotting the first 10 days of the temperature timeseries
`plt.plot(range(1440), temp[:1440])`

Figure 6.19 Temperature
over the first 10 days of the dataset (°C)

On this plot, you can see daily periodicity, especially evident for the last 4 days. Also note that this 10-day period must be coming from a fairly cold winter month.

If you were trying to predict average temperature for the next month given a few months of past data, the problem would be easy, due to the reliable yearscale periodicity of the data. But looking at the data over a scale of days, the temperature looks a lot more chaotic. Is this timeseries predictable at a daily scale? Let's find out.

Preparing the data

The exact formulation of the problem will be as follows: given data going as far back as `lookback` timesteps (a timestep is 10 minutes) and sampled every `steps` timesteps, can you predict the temperature in `delay` timesteps? You'll use the following parameter values:

`lookback = 720` —Observations will go back 5 days.

`steps = 6` —Observations will be sampled at one data point per hour. `delay = 144` —Targets will be 24 hours in the future.

To get started, you need to do two things:

Preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so you don't need to do any vectorization. But each timeseries in the data is on a different scale (for example, temperature is typically between -20 and +30, but atmospheric pressure, measured in mbar, is around 1,000). You'll normalize each timeseries independently so that they all take small values on a similar scale.

Write a Python generator that takes the current array of float data and yields batches of data from the recent past, along with a target temperature in the future. Because the samples in the dataset are highly redundant (sample N and sample $N + 1$ will have most of their timesteps in common), it would be wasteful to explicitly allocate every sample. Instead, you'll generate the samples on the fly using the original data.

You'll preprocess the data by subtracting the mean of each timeseries and dividing by the standard deviation. You're going to use the first 200,000 timesteps as training data, so compute the mean and standard deviation only on this fraction of the data.

Listing 6.32 Normalizing the data

```
mean = float_data[:200000].mean(axis=0) float_data -= mean
std = float_data[:200000].std(axis=0) float_data /= std
```

Listing 6.33 shows the data generator you'll use. It yields a tuple (samples, targets), where samples is one batch of input data and targets is the corresponding array of target temperatures. It takes the following arguments:

data—The original array of floating-point data, which you normalized in listing 6.32.

lookback—How many timesteps back the input data should go.

delay—How many timesteps in the future the target should be.

min_index and **max_index** —Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.

shuffle—Whether to shuffle the samples or draw them in chronological order.

batch_size—The number of samples per batch.

step—The period, in timesteps, at which you sample data. You'll set it to 6 in order to draw one data point every hour.

Listing 6.33 Generator yielding timeseries samples and their targets

```
def generator(data, lookback, delay, min_index, max_index, shuffle=False,
              batch_size=128, step=6):
```

if max_index is None:

```

max_index = len(data) - delay - 1
i = min_index + lookback while 1:
if shuffle:

rows = np.random.randint( min_index + lookback, max_index,
size=batch_size)

else:
if i + batch_size >= max_index:
i = min_index + lookback
rows = np.arange(i, min(i + batch_size, max_index)) i += len(rows) samples
= np.zeros((len(rows), lookback // step, data.shape[-1]))

targets = np.zeros((len(rows),)) for j, row in enumerate(rows): indices =
range(rows[j] - lookback, rows[j], step) samples[j] = data[indices] targets[j] =
data[rows[j] + delay][1]

yield samples, targets

```

Now, let's use the abstract generator function to instantiate three generators: one for training, one for validation, and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

Listing 6.34 Preparing the training, validation, and test generators lookback = 1440 step = 6 delay = 144 batch_size = 128
train_gen = generator(float_data,

```

lookback=lookback, delay=delay, min_index=0, max_index=200000,
shuffle=True, step=step,
batch_size=batch_size)

```

```

val_gen = generator(float_data, lookback=lookback, delay=delay,
min_index=200001,
max_index=300000, step=step, batch_size=batch_size) test_gen =
generator(float_data,
lookback=lookback, delay=delay, min_index=300001, max_index=None,

```

```
step=step, batch_size=batch_size) val_steps = (300000 200001 - lookback)
```

How many steps to draw from val_gen in order to see the entire validation set

How many steps to draw from test_gen in order to

```
test_steps = (len(float_data) - 300001 - lookback) see the entire test set
```

A common-sense, non-machine-learning baseline

Before you start using black-box deep-learning models to solve the temperature prediction problem, let's try a simple, common-sense approach. It will serve as a sanity check, and it will establish a baseline that you'll have to beat in order to demonstrate the usefulness of more-advanced machine learning models. Such common-sense baselines can be useful when you're approaching a new problem for which there is no known solution (yet). A classic example is that of unbalanced classification tasks, where some classes are much more common than others. If your dataset contains 90% instances of class A and 10% instances of class B, then a common-sense approach to the classification task is to always predict "A" when presented with a new sample. Such a classifier is 90% accurate overall, and any learning-based approach should therefore beat this 90% score in order to demonstrate usefulness. Sometimes, such elementary baselines can prove surprisingly hard to beat.

In this case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodical with a daily period. Thus a common-sense approach is to always predict that the temperature 24 hours from now will be equal to the temperature right now. Let's evaluate this approach, using the mean absolute error (MAE) metric:

```
np.mean(np.abs(preds - targets))
```

Here's the evaluation loop.

Listing 6.35 Computing the common-sense baseline MAE def evaluate_naive_method():


```

batch_maes = []
for step in range(val_steps):
    samples, targets = next(val_gen)
    preds = samples[:, -1, 1]
    mae = np.mean(np.abs(preds - targets))
    batch_maes.append(mae)
print(np.mean(batch_maes))
evaluate_naive_method()

```

This yields an MAE of 0.29. Because the temperature data has been normalized to be centered on 0 and have a standard deviation of 1, this number isn't immediately interpretable. It translates to an average absolute error of $0.29 \times \text{temperature\_stddegrees Celsius}$: 2.57°C.

Listing 6.36 Converting the MAE back to a Celsius error `celsius_mae = 0.29 * std[1]`

That's a fairly large average absolute error. Now the game is to use your knowledge of deep learning to do better.

A basic machine-learning approach

In the same way that it's useful to establish a common -sense baseline before trying machinelearning approaches, it's useful to try simple, cheap machine-learning models (such as small, densely connected networks) before looking into complicated and computationally expensive models such as RNNs. This is the best way to make sure any further complexity you throw at the problem is legitimate and delivers real benefits.

The following listing shows a fully connected model that starts by flattening the data and then runs it through two Dense layers. Note the lack of activation function on the last Dense layer, which is typical for a regression problem. You use MAE as the loss. Because you evaluate on the exact same data and with the exact same metric you did with the common-sense approach, the results will be directly comparable.

Listing 6.37 Training and evaluating a densely connected model from `keras.models` import `Sequential` from `keras` import `layers` from `keras.optimizers` import `RMSprop`

```

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback //
step, float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))

```

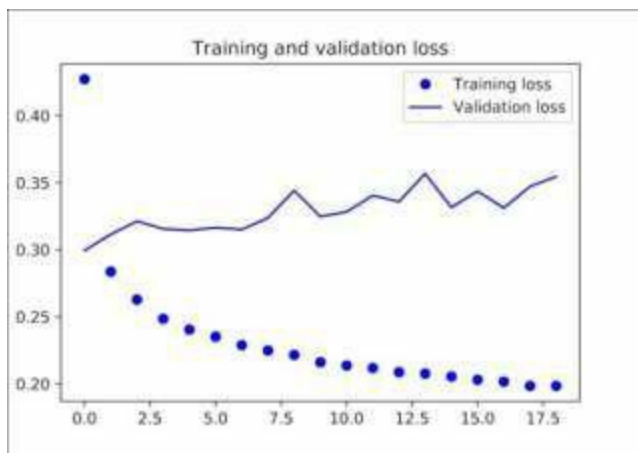
```
model.compile(optimizer=RMSprop(), loss='mae') history =  
model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,  
validation_data=val_gen, validation_steps=val_steps)
```

Let's display the loss curves for validation and training (see figure 6.20).

Plotting results

```
import matplotlib.pyplot as plt
```

```
loss = history.history['loss'] val_loss = history.history['val_loss'] epochs =  
range(1, len(loss) + 1) plt.figure()
```



```
plt.plot(epochs, loss, 'bo', label='Training loss')  
plt.plot(epochs, val_loss, 'b', label='Validation loss')  
plt.title("Training and validation loss") plt.legend() plt.show()
```

Figure 6.20 Training and validation loss on the Jena temperature- forecasting task with a simple, densely connected network

Some of the validation losses are close to the no-learning baseline, but not reliably. This goes to show the merit of having this baseline in the first place: it turns out to be not easy to outperform. Your common sense contains a lot of valuable information that a machinelearning model doesn't have access to.

You may wonder, if a simple, well-performing model exists to go from the data to the targets (the commonsense baseline), why doesn't the model you're

training find it and improve on it? Because this simple solution isn't what your training setup is looking for. The space of models in which you're searching for a solution—that is, your hypothesis space—is the space of all possible twolayer networks with the configuration you defined. These networks are already fairly complicated. When you're looking for a solution with a space of complicated models, the simple, well-performing baseline may be unlearnable, even if it's technically part of the hypothesis space. That is a pretty significant limitation of machine learning in general: unless the learning algorithm is hardcoded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem.

A first recurrent baseline

The first fully connected approach didn't do well, but that doesn't mean machine learning isn't applicable to this problem. The previous approach first flattened the timeseries, which removed the notion of time from the input data. Let's instead look at the data as what it is: a sequence, where causality and order matter. You'll try a recurrent-sequence processing model—it should be the perfect fit for such sequence data, precisely because it exploits the temporal ordering of data points, unlike the first approach.

Instead of the LSTM layer introduced in the previous section, you'll use the GRU layer, developed by Chung et al. in 2014.¹⁰ Gated recurrent unit (GRU) layers work using the same principle as LSTM, but they're somewhat streamlined and thus cheaper to run (although they may not have as much representational power as LSTM). This trade-off between computational expensiveness and representational power is seen everywhere in machine learning.

Training and evaluating a GRU-based model from keras.models import Sequential from keras import layers from keras.optimizers import RMSprop model = Sequential() model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1]))) model.add(layers.Dense(1))

```
model.compile(optimizer=RMSprop(), loss='mae') history =  
model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,  
validation_data=val_gen, validation_steps=val_steps)
```

Figure 6.21 shows the results. Much better! You can significantly beat the commonsense baseline, demonstrating the value of machine learning as well as the superiority of recurrent networks compared to sequence-flattening dense networks on this type of task.

validation loss on the

Jena temperature forecasting task with a GRU

The new validation MAE of ~ 0.265 (before you start significantly overfitting) translates to a mean absolute error of 2.35°C

after denormalization. That's a solid gain on the initial error of 2.57°C , but you probably still have a bit of a margin for improvement.

Using recurrent dropout to fight overfitting

It's evident from the training and validation curves that the model is overfitting: the training and validation losses start to diverge considerably after a few epochs. You're already familiar with a classic technique for fighting this phenomenon: dropout, which randomly zeros out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to. But how to correctly apply dropout in recurrent networks isn't a trivial question. It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. In 2015, Yarin Gal, as part of his PhD thesis on Bayesian deep learning,¹¹ determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that varies randomly from timestep to timestep. What's more, in order to regularize the representations formed by the recurrent gates of layers such as GRU and LSTM, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a recurrent dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.

Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers. Every recurrent layer in Keras has two

dropout-related arguments: dropout, a float specifying the dropout rate for input units of the layer, and recurrent_dropout, specifying the dropout rate of the recurrent units. Let's add dropout and recurrent dropout to the GRU layer and see how doing so impacts overfitting. Because networks being regularized with dropout always take longer to fully converge, you'll train the network for twice as many epochs.

```
Training and evaluating a dropout-regularized GRUbased model
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.GRU(32, dropout=0.2, recurrent_dropout=0.2, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
```

```
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500, epochs=40, validation_data=val_gen, validation_steps=val_steps)
```

Figure 6.22 shows the results. Success! You're no longer overfitting during the first 30 epochs. But although you have more stable evaluation scores, your best scores aren't much lower than they were previously. validation loss on the

Jena temperature forecasting task with a dropoutregularized GRU Stacking recurrent layers

Because you're no longer overfitting but seem to have hit a performance bottleneck,

you should consider increasing the capacity of the network. Recall the description of the universal machinelearning workflow: it's generally a good idea to increase the capacity of your network until overfitting becomes the primary obstacle (assuming you're already taking basic steps to mitigate overfitting, such as using dropout). As long as you aren't overfitting too badly, you're likely under capacity.

Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers. Recurrent layer stacking is a classic

way to build more-powerful recurrent networks: for instance, what currently powers the Google Translate algorithm is a stack of seven large LSTM layers—that's huge.

To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep. This is done by specifying `return_sequences=True`.

Training and evaluating a dropout-regularized, stacked GRU model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
```

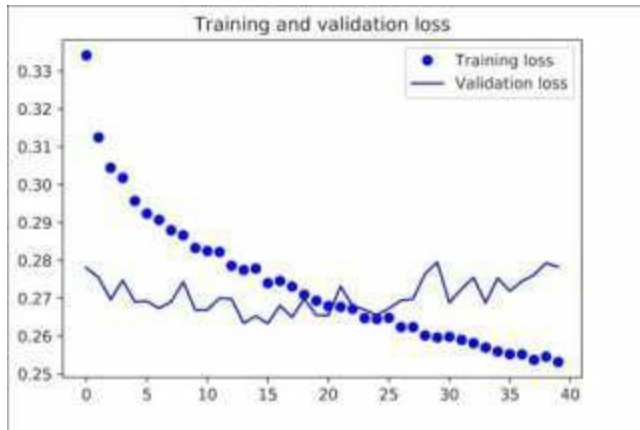
```
model = Sequential()
model.add(layers.GRU(32, dropout=0.1,
                    recurrent_dropout=0.5, return_sequences=True,
                    input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                    dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))
```

```
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500, epochs=40,
                             validation_data=val_gen, validation_steps=val_steps)
```

Figure 6.23 shows the results. You can see that the added layer does improve the results a bit, though not significantly. You can draw two conclusions:

the

a



help

Because you're still not overfitting too badly, you could safely increase size of your layers in a quest for validation-loss improvement. This has non-negligible computational cost, though.

Adding a layer didn't by a significant factor, so you may be seeing diminishing returns from increasing network capacity at this point.

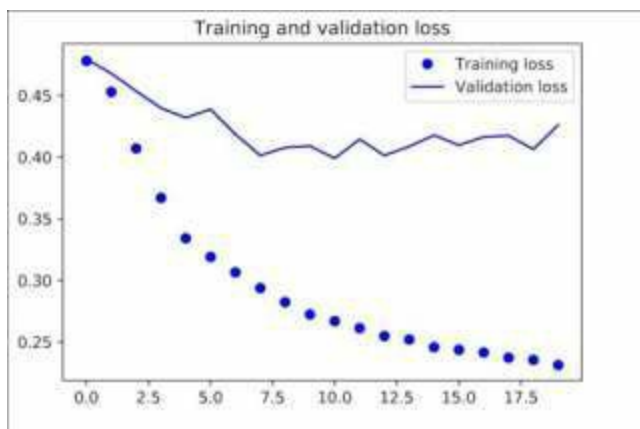
Figure 6.23 Training and validation loss on the Jena temperature forecasting task with a stacked GRU network
Using bidirectional RNNs

The last technique introduced in this section is called bidirectional RNNs. A bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks. It's frequently used in natural-language processing—you could call it the Swiss Army knife of deep learning for natural-language processing.

RNNs are notably order dependent, or time dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations the RNN extracts from the sequence. This is precisely the reason they perform well on problems where order is meaningful, such as the temperature-forecasting problem. A bidirectional RNN exploits the order sensitivity of RNNs: it consists of using

two regular RNNs, such as the GRU and LSTM layers you're already familiar with, each of which processes the input sequence in one direction (chronologically and antichronologically), and then merging their representations. By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.

Remarkably, the fact that the RNN layers in this section have processed sequences in chronological order (older timesteps first) may have been an arbitrary decision. At least, it's a decision we made no attempt to question so far. Could the RNNs have performed well enough if they processed input sequences in antichronological order, for instance (newer timesteps first)?



Let's try this in practice and see what happens. All you need to do is write a variant of the data generator where the input sequences are reverted along the time dimension (replace the last line with `yield samples[:, ::-1, :], targets`). Training the same one-GRU layer network that you used in the first experiment in this

section, you get the results shown in figure 6.24.

Figure 6.24 Training and validation loss on the Jena temperature forecasting task with a GRU trained on reversed sequences

The reversed-order GRU strongly underperforms even the common-sense baseline, indicating that in this case, chronological processing is important to the success of your approach. This makes perfect sense: the underlying GRU layer will typically be better at remembering the recent past than the distant past, and naturally the more recent weather data points are more predictive

than older data points for the problem (that's what makes the common-sense baseline fairly strong). Thus the chronological version of the layer is bound to outperform the reversedorder version. Importantly, this isn't true for many other problems, including natural language: intuitively, the importance of a word in understanding a sentence isn't usually dependent on its position in the sentence. Let's try the same trick on the LSTM IMDB example from section 6.2.

Listing 6.42 Training and evaluating an **LSTM** using reversed sequences

```
from keras.datasets import imdb from
keras.preprocessing import sequence from keras import layers

from keras.models import Sequential Number of words

to consider as
max_features = 10000 features
maxlen = 500

(x_train, y_train), (x_test, y_test) =
imdb.load_data(
num_words=max_features)

Loads
data x_train = [x[::-1] for x in x_train]Reverses x_test = [x[::-1] for x in
x_test]sequences

x_train = sequence.pad_sequences(x_train,
maxlen=maxlen) x_test =
sequence.pad_sequences(x_test, maxlen=maxlen) model =
Sequential()
model.add(layers.Embedding(max_features, 128)) model.add(layers
s.LSTM(32))
model.add(layers.Dense(1,
activation='sigmoid'))

model.compile(optimizer='rmsprop',
loss='binary_crossentropy',
```

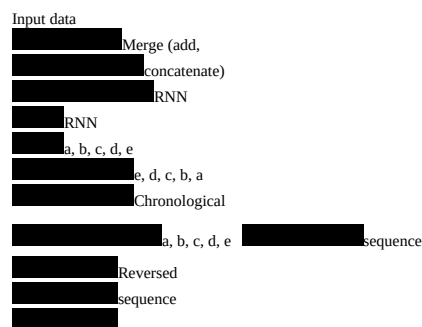
```
metrics=['acc'])
```

Cuts off texts after this number of words (among the max_features most common words)

Pads sequences

```
history = model.fit(x_train, y_train,  
epochs=10, batch_size=128, validation_split=0.2)
```

You get performance nearly identical to that of the chronological-order LSTM. Remarkably, on such a text dataset, reversed-order processing works just as well as chronological processing, confirming the hypothesis that, although word order does matter in understanding language, which order you use isn't crucial. Importantly, an RNN trained on reversed sequences will learn different representations than one trained on the original sequences, much as you would have different mental models if time flowed backward in the real world— if you lived a life where you died on your first day and were born on your last day. In machine learning, representations that are different yet useful are always worth exploiting, and the more they differ, the better: they offer a new angle from which to look at your data, capturing aspects of the data that were



missed by other approaches, and thus they can help boost performance on a task. This is the intuition behind ensembling, a concept we'll explore in chapter 7.

A bidirectional RNN exploits this idea to improve on the performance of chronologicalorder RNNs. It looks at its input sequence both ways (see figure

6.25), obtaining potentially richer representations and

capturing patterns that may have been missed by the chronological-order version alone.

Figure 6.25 How a bidirectional RNN layer works

To instantiate a bidirectional RNN in Keras, you use the `Bidirectional` layer, which takes as its first argument a recurrent layer instance. `Bidirectional` creates a second, separate instance of this recurrent layer and uses one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order. Let's try it on the IMDB sentiment-analysis task.

Training and evaluating a bidirectional **LSTM**

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train,
y_train,
```

```
epochs=10, batch_size=128, validation_split=0.2)
```

It performs slightly better than the regular LSTM you tried in the previous section, achieving over 89% validation accuracy. It also seems to overfit more quickly, which is unsurprising because a bidirectional layer has twice as many parameters as a chronological LSTM. With some regularization, the bidirectional approach would likely be a strong performer on this task.

Now let's try the same approach on the temperature-prediction task.

Training a bidirectional **GRU**

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Bidirectional(layers.GRU(32),
input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
```

```
model.compile(optimizer=RMSprop(), loss='mae') history =  
model.fit_generator(train_gen, steps_per_epoch=500, epochs=40,  
validation_data=val_gen, validation_steps=val_steps)
```

This performs about as well as the regular GRU layer. It's easy to understand why: all the predictive capacity must come from the chronological half of the network, because the antichronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case).

Going even further

There are many other things you could try, in order to improve performance on the temperature-forecasting problem:

Adjust the number of units in each recurrent layer in the stacked setup. The current choices are largely arbitrary and thus probably suboptimal.

Adjust the learning rate used by the RMSprop optimizer.
Try using LSTM layers instead of GRU layers.

Try using a bigger densely connected regressor on top of the recurrent layers: that is, a bigger Dense layer or even a stack of Dense layers.

Don't forget to eventually run the best-performing models (in terms of validation MAE) on the test set! Otherwise, you'll develop architectures that are overfitting to the validation set.

As always, deep learning is more an art than a science. We can provide guidelines that suggest what is likely to work or not work on a given problem, but, ultimately, every problem is unique; you'll have to evaluate different strategies empirically. There is currently no theory that will tell you in advance precisely what you should do to optimally solve a problem. You must iterate.

Wrapping up

Here's what you should take away from this section:

As you first learned in chapter 4, when approaching a new problem, it's good

to first establish common-sense baselines for your metric of choice. If you don't have a baseline to beat, you can't tell whether you're making real progress.

Try simple models before expensive ones, to justify the additional expense. Sometimes a simple model will turn out to be your best option.

When you have data where temporal ordering matters, recurrent networks are a great fit and easily outperform models that first flatten the temporal data.

To use dropout with recurrent networks, you should use a time-constant dropout mask and recurrent dropout mask. These are built into Keras recurrent layers, so all you have to do is use the dropout and recurrent_dropout arguments of recurrent layers.

Stacked RNNs provide more representational power than a single RNN layer. They're also much more expensive and thus not always worth it. Although they offer clear gains on complex problems (such as machine translation), they may not always be relevant to smaller, simpler problems.

Bidirectional RNNs, which look at a sequence both ways, are useful on natural language processing problems. But they aren't strong performers on sequence data where the recent past is much more informative than the beginning of the sequence.

NOTE There are two important concepts we won't cover in detail here: recurrent attention and sequence masking. Both tend to be especially relevant for natural language processing, and they aren't particularly applicable to the temperature forecasting problem. We'll leave them for future study outside of this book.

Markets and machine learning

Some readers are bound to want to take the techniques we've introduced here and try them on the problem of forecasting the future price of securities on the stock market (or currency exchange rates, and so on). Markets have very different statistical characteristics than natural phenomena such as weather patterns. Trying to use machine learning to beat markets, when you only have access to publicly available data, is a difficult endeavor, and you're likely to waste your time and resources with nothing to show for it.

Always remember that when it comes to markets, past performance is not a good predictor of future returns—looking in the rear-view mirror is a bad way to drive. Machine learning, on the other hand, is applicable to datasets where the past is a good predictor of the future.

Sequence processing with convnets

you learned about convolutional neural networks (convnets) and how they perform particularly well on computer vision problems, due to their ability to operate convolutionally, extracting features from local input patches and allowing for representation modularity and data efficiency. The same properties that make convnets excel at computer vision also make them highly relevant to sequence processing. Time can be treated as a spatial dimension, like the height or width of a 2D image.

Such 1D convnets can be competitive with RNNs on certain sequence processing problems, usually at a considerably cheaper computational cost. Recently, 1D convnets, typically used with dilated kernels, have been used with great success for audio generation and machine translation. In addition to these specific successes, it has long been known that small 1D convnets can offer a fast alternative to RNNs for simple tasks such as text classification and timeseries forecasting.

Understanding 1D convolution for sequence data

The convolution layers introduced previously were 2D convolutions, extracting 2D

patches from image tensors

and applying an identical window of size 5

transformation to every patch.

In

the same way, you can use 1D convolutions, extracting local 1D patches (subsequences) from

Time

sequences (see figure 6.26). Input ^{Extracted}

features_{patch}

Dot product

Figure 6.26 How 1D

with weights

convolution Output works: each output timestep is features

Output

obtained from a temporal patch in the input sequence.

Such 1D convolution layers can recognize local patterns in a sequence. Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position, making 1D convnets translation invariant (for temporal translations). For instance, a 1D convnet processing sequences of characters using convolution windows of size 5 should be able to learn words or word fragments of length 5 or less, and it should be able to recognize these words in any context in an input sequence. A character-level 1D convnet is thus able to learn about word morphology.

1D pooling for sequence data

You're already familiar with 2D pooling operations, such as 2D average pooling and max pooling, used in convnets to spatially downsample image tensors. The 2D pooling operation has a 1D equivalent: extracting 1D patches (subsequences) from an input and outputting the maximum value (max pooling) or average value (average pooling). Just as with 2D convnets, this is used for reducing the length of 1D inputs (subsampling).

Implementing a 1D convnet

In Keras, you use a 1D convnet via the Conv1D layer, which has an interface similar to

Conv2D. It takes as input 3D tensors with shape (samples, time, features) and returns similarly shaped 3D tensors. The convolution window is a 1D window on the temporal axis: axis 1 in the input tensor.

Let's build a simple two-layer 1D convnet and apply it to the IMDB sentimentclassification task you're already familiar with. As a reminder, this is the code for obtaining and preprocessing the data.

Listing 6.45 Preparing the IMDB data

```
from keras.datasets import imdb from keras.preprocessing import sequence
max_features = 10000 max_len = 500
print('Loading data...')
```

```
(x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=max_features) print(len(x_train), 'train
sequences') print(len(x_test), 'test sequences')
print('Pad sequences (samples x time)') x_train =
sequence.pad_sequences(x_train, maxlen=max_len) x_test =
sequence.pad_sequences(x_test, maxlen=max_len) print('x_train shape:',
x_train.shape) print('x_test shape:', x_test.shape)
```

1D convnets are structured in the same way as their 2D counterparts, which you used in chapter 5: they consist of a stack of Conv1D and MaxPooling1D layers, ending in either a global pooling layer or a Flatten layer, that turn the 3D outputs into 2D outputs, allowing you to add one or more Dense layers to the model for classification or regression.

One difference, though, is the fact that you can afford to use larger convolution windows with 1D convnets. With a 2D convolution layer, a 3×3 convolution window contains $3 \times 3 = 9$ feature vectors; but with a 1D convolution layer, a convolution window of size 3 contains only 3 feature vectors. You can thus easily afford 1D convolution windows of size 7 or 9.

This is the example 1D convnet for the IMDB dataset.

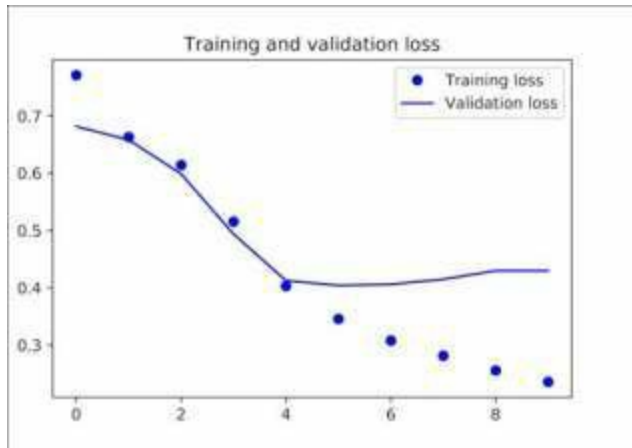
```
Training and evaluating a simple 1D convnet on the IMDB data
from keras.models import Sequential from keras import layers from
keras.optimizers import RMSprop
model = Sequential()
```

```
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5)) model.add(layers.Conv1D(32, 7,
activation='relu')) model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1)) model.summary()
```

```
model.compile(optimizer=RMSprop(lr=1e-4), loss='binary_crossentropy',
```

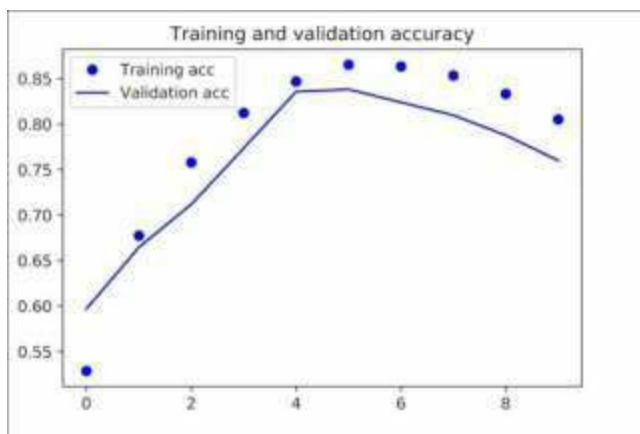


```
metrics=['acc'])
history = model.fit(x_train, y_train,
epochs=10, batch_size=128, validation_split=0.2)
```



Figures 6.27 and 6.28 show the training and validation results. Validation accuracy is somewhat less than that of the LSTM, but runtime is faster on both CPU and GPU (the exact increase in speed will vary greatly depending on your exact configuration). At this point, you could retrain this model for the right number of

on



and
with a simple 1D convnet
epochs (eight) and run it the test set. This is a convincing

demonstration that a 1D convnet can offer a fast, cheap alternative to a recurrent network on a word-level sentimentclassification task.

Figure 6.27 Training validation loss on IMDB

Figure 6.28 Training and validation accuracy on IMDB with a simple 1D convnet

Combining CNNs and RNNs to process long sequences

Because 1D convnets process input patches independently, they aren't sensitive to the order of the timesteps (beyond a local scale, the size of the convolution windows), unlike RNNs. Of course, to recognize longer-term patterns, you can stack many convolution layers and pooling layers, resulting in upper layers that will see long chunks of the original inputs—but that's still a fairly weak way to induce order sensitivity. One way to evidence this weakness is to try 1D convnets on the temperature-forecasting problem, where order-sensitivity is key to producing good predictions. The following example reuses the following variables defined previously: `float_data`, `train_gen`, `val_gen`, and `val_steps`.

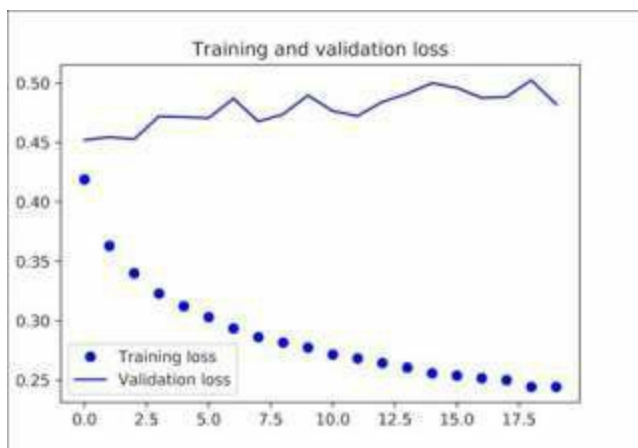
Training and evaluating a simple 1D convnet on the Jena data

```
from keras.models import Sequential from keras import layers from
```

```
keras.optimizers import RMSprop
```

```
model = Sequential() model.add(layers.Conv1D(32, 5, activation='relu',  
input_shape=(None, float_data.shape[-1])))
```

```
model.add(layers.MaxPooling1D(3)) model.add(layers.Conv1D(32, 5,  
activation='relu')) model.add(layers.MaxPooling1D(3))
```



```
model.add(layers.Conv1D(32,  
5, activation='relu'))
```

```
model.add(layers.GlobalMaxPooling1D()) model.add(layers.Dense(1))
```

```
model.compile(optimizer=RMSprop(), loss='mae') history =  
model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,  
validation_data=val_gen, validation_steps=val_steps)
```

Figure 6.29 shows the training and validation MAEs.

Figure 6.29 Training and validation loss on the Jena temperature-forecasting task with a simple 1D convnet

The validation MAE stays in the 0.40s: you can't even beat the common -sense baseline using the small convnet. Again, this is because the convnet looks for patterns anywhere in the input timeseries and has no knowledge of the temporal position of a pattern it sees (toward the beginning, toward the end, and so on). Because more recent data points should be interpreted differently from older data points in the case of this specific forecasting problem, the convnet fails at producing meaningful results. This limitation of convnets isn't an issue with the IMDB data, because patterns of keywords associated with a positive or negative sentiment are informative independently of where they're found in the input sentences.

One strategy to combine the speed and lightness of convnets with the ordersensitivity of RNNs is to use a 1D convnet as a preprocessing step before an RNN (see figure 6.30).

This is especially beneficial when you're dealing with sequences that are so long they can't realistically be processed with RNNs, such as sequences with thousands of steps. The convnet will turn the long input sequence into much shorter (downsampled) sequences of higher-level features. This sequence of extracted features then becomes

the input to the RNN part of the network.

This technique isn't seen often in research papers and practical applications, possibly because it isn't well known. It's effective and ought to be more common. Let's try it on the temperature-forecasting dataset. Because this strategy allows you to manipulate much longer sequences, you can either

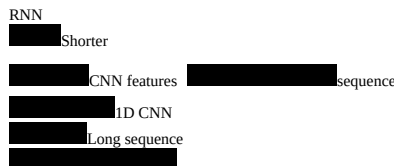


Figure 6.30 Combining a 1D convnet and an RNN for processing long sequences

look at data from longer ago (by increasing the lookback parameter of the data generator) or look at high-resolution timeseries (by decreasing the step parameter of the generator). Here, somewhat arbitrarily, you'll use a step that's half as large, resulting in a timeseries twice as long, where the temperature data is sampled at a rate of 1 point per 30 minutes. The example reuses the generator function defined earlier.

Preparing higher-resolution data generators for the Jena dataset

step = 3

Previously lookback = 720

set to 6 (1 point per hour); **Unchanged now 3 (1 point per 30 min)** delay = 144

```
train_gen = generator(float_data, lookback=lookback, delay=delay,
min_index=0, max_index=200000, shuffle=True, step=step)
val_gen = generator(float_data, lookback=lookback, delay=delay,
min_index=200001, max_index=300000, step=step)
```

```
test_gen = generator(float_data, lookback=lookback, delay=delay,
```

```
min_index=300001, max_index=None, step=step)
val_steps = (300000 - 200001 - lookback) // 128 test_steps = (len(float_data)
300001 - lookback) // 128
```

This is the model, starting with two Conv1D layers and following up with a GRU layer. Figure 6.31 shows the results.

```
Model combining a 1D convolutional base and a GRU layer
from keras.models import Sequential from keras import layers from
keras.optimizers import RMSprop
model = Sequential() model.add(layers.Conv1D(32, 5, activation='relu',
input_shape=(None, float_data.shape[-1])))
```



```
model.add(layers.MaxPooling1D(3)) model.add(layers.Conv1D(32, 5,
activation='relu')) model.add(layers.GRU(32, dropout=0.1,
recurrent_dropout=0.5)) model.add(layers.Dense(1)) model.summary()
model.compile(optimizer=RMSprop(), loss='mae')
```

```
history = model.fit_generator(train_gen, steps_per_epoch=500, epochs=20,
validation_data=val_gen, validation_steps=val_steps)
```

Figure 6.31 Training and validation loss on the Jena temperature forecasting task with a 1D convnet followed by a **GRU**

Judging from the validation loss, this setup isn't as good as the regularized GRU alone, but it's significantly faster. It looks at twice as much data, which in this case doesn't appear to be hugely helpful but may be important for other datasets. Wrapping up

Here's what you should take away from this section:

In the same way that 2D convnets perform well for processing visual patterns in 2D space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular natural language processing tasks.

Typically, 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of Conv1D layers and MaxPooling1D layers, ending in a global pooling operation or flattening operation.

Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before an RNN, shortening the sequence and extracting useful representations for the RNN to process.

Chapter summary

In this chapter, you learned the following techniques, which are widely applicable to any dataset of sequence data, from text to timeseries:

How to tokenize text

What word embeddings are, and how to use them

What recurrent networks are, and how to use them

How to stack RNN layers and use bidirectional RNNs to build more-powerful sequence-processing models

How to use 1D convnets for sequence processing

How to combine 1D convnets and RNNs to process long sequences

You can use RNNs for timeseries regression (“predicting the future”), timeseries classification, anomaly detection in timeseries, and sequence labeling (such as identifying names or dates in sentences).

Similarly, you can use 1D convnets for machine translation (sequence-to-sequence convolutional models, like SliceNet^a), document classification, and spelling correction.

If global order matters in your sequence data, then it's preferable to use a recurrent network to process it. This is typically the case for timeseries, where the recent past is likely to be more informative than the distant past.

If global ordering isn't fundamentally meaningful, then 1D convnets will turn out to work at least as well and are cheaper. This is often the case for text data, where a keyword found at the beginning of a sentence is just as meaningful as a keyword found at the end.

^a See

5 DEEP LEARNING ADVANCED API

This chapter covers
The Keras functional API
Using Keras callbacks
Working with the TensorBoard visualization tool
Important best practices for developing state-of-the-art models

This chapter explores a number of powerful tools that will bring you closer to being able to develop state-of-the-art models on difficult problems. Using the Keras functional API, you can build graph-like models, share a layer across different inputs, and use Keras models just like Python functions. Keras callbacks and the TensorBoard browser-based visualization tool let you monitor models during training. We'll also discuss several other best practices including batch normalization, residual connections, hyperparameter optimization, and model ensembling.

233

Going beyond the Sequential model:
the Keras functional API

Until now, all neural networks introduced in this book have been implemented using the Sequential model. The Sequential model makes the assumption that the network has exactly one input and exactly one output, and that it consists of a linear stack of layers (see figure 7.1).

This is a commonly verified assumption; the configuration is so common that we've been able to cover many topics and practical applications in these pages so far using only the Sequential model class. But this set of assumptions is too inflexible in a number of cases. Some networks require several independent inputs, others require multiple outputs, and some networks have internal branching between layers that makes them look like graphs of layers rather than linear stacks of layers.

Output
Layer
Layer

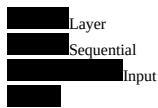


Figure 7.1 A **Sequential** model: a linear stack of layers

Some tasks, for instance, require multimodal inputs: they merge data coming from different input sources, processing each type of data using different kinds of neural layers. Imagine a deep-learning model trying to predict the most likely market price of a second-hand piece of clothing, using the following inputs: userprovided metadata (such as the item’s brand, age, and so on), a user-provided text description, and a picture of the item. If you had only the metadata available, you could one-hot encode it and use a densely connected network to predict the price. If you had only the text description available, you could use an RNN or a 1D convnet. If you had only the picture, you could use a 2D convnet. But how can you use all three at the same time? A naive approach would be to train three separate models and then do a weighted average of their predictions. But this may be suboptimal, because the information extracted by the models may be redundant. A better way is to jointly learn a more accurate model of the data by using a model that can see all available input modalities simultaneously: a model with three input branches (see figure 7.2).

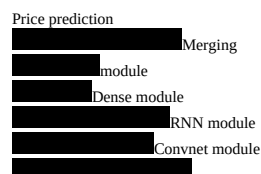


Figure 7.2 A multiinput model

Similarly, some tasks need to predict multiple target attributes of input data. Given the text of a novel or short story, you might want to automatically classify it by genre (such as romance or thriller) but also predict the approximate date it was written. Of course, you could train two separate models: one for the genre and one for the date. But because these attributes aren’t statistically independent, you could build a better model by learning to jointly predict both genre and date at the same time. Such a joint model would then have two outputs, or heads (see figure 7.3). Due to correlations between genre and date, knowing the date of a novel would help the model learn rich, accurate representations of the space of novel genres, and vice

versa.

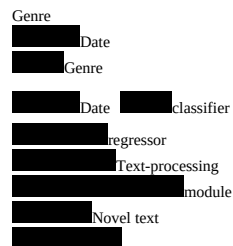


Figure 7.3 A multi-output (or multihead) model

Additionally, many recently developed neural architectures require nonlinear network topology: networks structured as directed acyclic graphs. The Inception family of networks (developed by Szegedy et al. at Google),¹² for instance, relies on Inception modules, where the input is processed by several parallel convolutional branches whose outputs are then merged back into a single tensor (see figure 7.4). There’s also the recent trend of adding residual connections to a model, which started with the ResNet family of networks (developed by He et al. at Microsoft).¹³ A residual connection consists of reinjecting previous representations into the downstream flow of data by adding a past output tensor to a later output tensor (see figure 7.5), which helps prevent information loss along the data-processing flow. There are many other examples of such graph-like networks.

¹²Christian Szegedy et al., “Going Deeper with Convolutions,” Conference on Computer Vision and Pattern Recognition (2014), <https://arxiv.org/abs/1409.4842>.

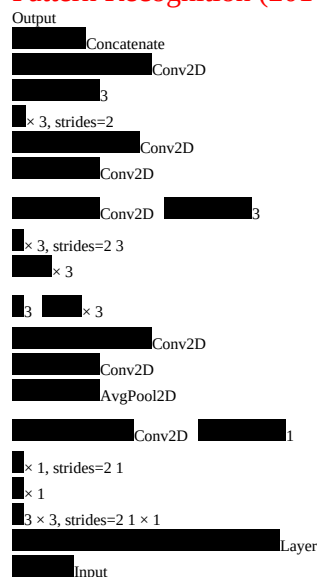


Figure 7.4 An Inception module: a subgraph of

■ layers with several parallel convolutional branches

Layer

■ Residual connection

Layer

■ Figure 7.5 A residual connection: reinjection of prior information downstream via feature-map addition

Layer

■

These three important use cases —multi-input models, multi-output models, and graph-like models—aren’t possible when using only the `Sequential` model class in Keras. But there’s another far more general and flexible way to use Keras: the functional API. This section explains in detail what it is, what it can do, and how to use it.

¹³Kaiming He et al., “Deep Residual Learning for Image Recognition,” *Conference on Computer Vision and Pattern Recognition* (2015), <https://arxiv.org/abs/1512.03385>.

Introduction to the functional API

In the functional API, you directly manipulate tensors, and you use layers as functions that take tensors and return tensors (hence, the name functional API):

```
from keras import Input, layers
input_tensor = Input(shape=(32,))
dense = layers.Dense(32, activation='relu')
output_tensor = dense(input_tensor)
```

A tensor
A layer is a function.

A layer may be called on a tensor, and it returns a tensor. Let’s start with a minimal example that shows side by side a simple `Sequential` model and its equivalent in the functional API:

```
from keras.models import Sequential, Model
from keras import layers
from keras import Input
```

Sequential model, which you already know about

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu',
input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

```
input_tensor = Input(shape=(64,))
```

```
x = layers.Dense(32,
activation='relu')(input_tensor) Its functional x = layers.Dense(32,
activation='relu')(x) equivalent
```

```
output_tensor = layers.Dense(10, activation='softmax')(x) model =
Model(input_tensor, output_tensor)
The Model class turns an input tensor
model.summary() Let's look at it! and output tensor into a model.
```

This is what the call to model.summary() displays:

```
_____
_____

Layer (type) Output Shape Param #
=====
=====
input_1 (InputLayer)(None, 64) 0

_____

dense_1 (Dense) (None, 32) 2080

_____

dense_2 (Dense) (None, 32) 1056

_____

dense_3 (Dense) (None, 10) 330
=====
=====

Total params: 3,466
Trainable params: 3,466 Non-trainable params: 0

_____
_____
```

The only part that may seem a bit magical at this point is instantiating a Model object using only an input tensor and an output tensor. Behind the scenes, Keras retrieves every layer involved in going from input_tensor to

output_tensor, bringing them together into a graph-like data structure—a Model. Of course, the reason it works is that output_tensor was obtained by repeatedly transforming input_tensor. If you tried to build a model from inputs and outputs that weren't related, you'd get a RuntimeError:

```
>>> unrelated_input = Input(shape=(32,))
```

```
>>> bad_model = model = Model(unrelated_input, output_tensor)
```

```
RuntimeError: Graph disconnected: cannot obtain value for tensor
```

```
↳ Tensor("input_1:0", shape=(?, 64), dtype=float32) at layer "input_1".
```

This error tells you, in essence, that Keras couldn't reach input_1 from the provided output tensor.

When it comes to compiling, training, or evaluating such an instance of Model, the API is the same as that of Sequential:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Compiles

```
import numpy as np
```

Generates dummy Numpy

```
x_train = np.random.random((1000, 64))
```

```
y_train = np.random.random((1000, 10))
```

```
model.fit(x_train, y_train,
```

```
epochs=10, batch_size=128)
```

Trains the model

```
score = model.evaluate(x_train, y_train)
```

for 10 epochs

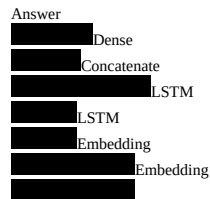
Evaluates the model

Multi-input models

The functional API can be used to build models that have multiple inputs. Typically, such models at some point merge their different input branches using a layer that can combine several tensors: by adding them, concatenating them, and so on. This is usually done via a Keras merge operation such as keras.layers.add, keras.layers.concatenate, and so on. Let's look at a very simple example of a multi-input model: a question-answering model.

A typical question-answering model has two inputs: a natural-language

question and a text snippet (such as a news article) providing information to be used for answering the question. The model must then produce an answer: in the simplest possible setup, this is a one-word answer obtained via a softmax over some predefined vocabulary (see figure 7.6).



Reference text Question Figure 7.6 A question-answering model

Following is an example of how you can build such a model with the functional API. You set up two independent branches, encoding the text input and the question input as representation vectors; then, concatenate these vectors; and finally, add a softmax classifier on top of the concatenated representations.

Functional API implementation of a two-input questionanswering model
from keras.models import Model from keras import layers from keras import Input

The text input is a variable

text_vocabulary_size = 10000 **length sequence of integers.**

Note that you can optionally

question_vocabulary_size = 10000

name the inputs.

answer_vocabulary_size = 500 text_input = Input(shape=(None,),
dtype='int32', name='text')

embedded_text = layers.Embedding(64, text_vocabulary_size)(text_input)

Embeds the inputs into a sequence of vectors of size 64 encoded_text =

layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,), **Encodes the vectors in a**

dtype='int32', **single vector via an LSTM**

name='question')

Same process (with different layer instances) for the question

embedded_question = layers.Embedding(

```
32, question_vocabulary_size)(question_input) encoded_question =  
layers.LSTM(16)(embedded_question)
```

```
concatenated = layers.concatenate([encoded_text, encoded_question],  
axis=-1)
```

Concatenates the encoded

```
answer = layers.Dense(answer_vocabulary_size, question and encoded text
```

```
activation='softmax')(concatenated)
```

Adds a softmax

```
model = Model([text_input, question_input], answer) classifier on top  
model.compile(optimizer='rmsprop',
```

At model instantiation, you specify

```
loss='categorical_crossentropy',
```

the two inputs and the output.

```
metrics=['acc'])
```

Now, how do you train this two-input model? There are two possible APIs: you can feed the model a list of Numpy arrays as inputs, or you can feed it a dictionary that maps input names to Numpy arrays. Naturally, the latter option is available only if you give names to your inputs.

Feeding data to a multi-input model

```
import numpy as np Generates dummy
```

Numpy data

```
num_samples = 1000 max_length = 100
```

```
text = np.random.randint(1, text_vocabulary_size,  
size=(num_samples, max_length))
```

Answers are one

```
question = np.random.randint(1, question_vocabulary_size, hot encoded,  
size=(num_samples, max_length)) not integers
```

```
answers = np.random.randint(0, 1, size=(num_samples,  
answer_vocabulary_size))
```

```
model.fit([text, question], answers, epochs=10, batch_size=128)
```

```
model.fit({'text': text, 'question': question}, answers, epochs=10,
batch_size=128)
```

Fitting using a list of inputs Fitting using a dictionary of inputs (only if inputs are named)

Multi-output models

In the same way, you can use the functional API to build models with multiple outputs (or multiple heads). A simple example is a network that attempts to simultaneously predict different properties of the data, such as a network that takes as input a series of social media posts from a single anonymous person and tries to predict attributes of that person, such as age, gender, and income level (see figure 7.7).

Listing 7.3 Functional API implementation of a three-output model

```
from keras import layers from keras import Input from keras.models import
Model
```

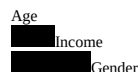
```
vocabulary_size = 50000 num_income_groups = 10
```

```
posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input) x =
layers.Conv1D(128, 5, activation='relu')(embedded_posts) x =
layers.MaxPooling1D(5)(x) x = layers.Conv1D(256, 5, activation='relu')(x) x
= layers.Conv1D(256, 5, activation='relu')(x) x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x) x = layers.Conv1D(256, 5,
activation='relu')(x) x = layers.GlobalMaxPooling1D()(x)
```

```
x = layers.Dense(128, activation='relu')(x) Note that the output layers are
given names.
```

```
age_prediction = layers.Dense(1, name='age')(x) income_prediction =
layers.Dense(num_income_groups, activation='softmax', name='income')(x)
```

```
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)
model = Model(posts_input,
[age_prediction, income_prediction, gender_prediction])
```



Age
Income
Gender

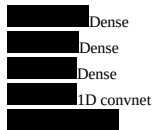


Figure 7.7 A social media
Social media posts model with three heads

Importantly, training such a model requires the ability to specify different loss functions for different heads of the network: for instance, age prediction is a scalar regression task, but gender prediction is a binary classification task, requiring a different training procedure. But because gradient descent requires you to minimize a scalar, you must combine these losses into a single value in order to train the model. The simplest way to combine different losses is to sum them all. In Keras, you can use either a list or a dictionary of losses in compile to specify different objects for different outputs; the resulting loss values are summed into a global loss, which is minimized during training.

Compilation options of a multi-output model: multiple losses

```
model.compile(optimizer='rmsprop',
loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'])
```

model.compile(optimizer='rmsprop', **Equivalent (possible** loss={'age': 'mse', **only**

if you give names
'income': 'categorical_crossentropy', **to the output layers)**
'gender': 'binary_crossentropy'})

Note that very imbalanced loss contributions will cause the model representations to be optimized preferentially for the task with the largest individual loss, at the expense of the other tasks. To remedy this, you can assign different levels of importance to the loss values in their contribution to the final loss. This is useful in particular if the losses' values use different scales. For instance, the mean squared error (MSE) loss used for the age regression task typically takes a value around 3–5, whereas the crossentropy loss used for the gender-classification task can be as low as 0.1. In such a situation, to balance the contribution of the different losses, you can assign a weight of 10 to the crossentropy loss and a weight of 0.25 to the

MSE loss.

Compilation options of a multi-output model: loss weighting

```
model.compile(optimizer='rmsprop', loss=['mse', 'categorical_crossentropy',  
'binary_crossentropy'], loss_weights=[0.25, 1., 10.]
```

```
model.compile(optimizer='rmsprop',  
loss={'age': 'mse',  
'income': 'categorical_crossentropy'}, Equivalent (possible
```

```
'gender': 'binary_crossentropy'}), only if you give names loss_weights={'age':  
0.25, to the output layers)
```

```
'income': 1.,  
'gender': 10.]
```

Much as in the case of multi-input models, you can pass Numpy data to the model for training either via a list of arrays or via a dictionary of arrays.

Feeding data to a multi-output model

```
model.fit(posts, [age_targets, income_targets, gender_targets], epochs=10,  
batch_size=64)
```

```
model.fit(posts, {'age': age_targets,  
'income': income_targets}, Equivalent (possible only if you
```

```
'gender': gender_targets}), give names to the output layers) epochs=10,  
batch_size=64)
```

age_targets, income_targets, and gender_targets are assumed to be Numpy arrays.

Directed acyclic graphs of layers

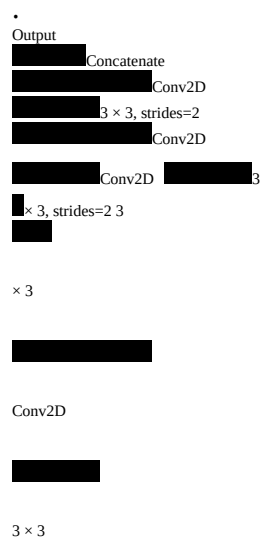
With the functional API, not only can you build models with multiple inputs and multiple outputs, but you can also implement networks with a complex internal topology. Neural networks in Keras are allowed to be arbitrary directed acyclic graphs of layers. The qualifier acyclic is important: these graphs can't have cycles. It's impossible for a tensor x to become the input of one of the layers that generated x . The only processing loops that are allowed

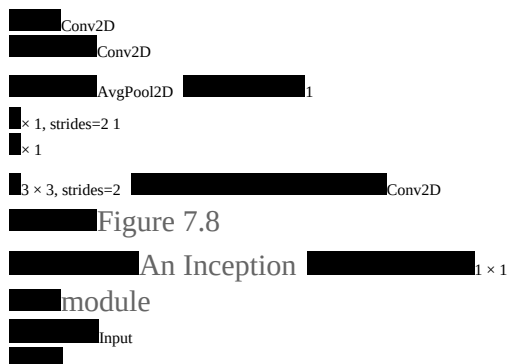
(that is, recurrent connections) are those internal to recurrent layers.

Several common neural-network components are implemented as graphs. Two notable ones are Inception modules and residual connections. To better understand how the functional API can be used to build graphs of layers, let's take a look at how you can implement both of them in Keras.

INCEPTION MODULES

Inception¹⁴ is a popular type of network architecture for convolutional neural networks; it was developed by Christian Szegedy and his colleagues at Google in 2013–2014, inspired by the earlier network-in-network architecture.¹⁵ It consists of a stack of modules that themselves look like small independent networks, split into several parallel branches. The most basic form of an Inception module has three to four branches starting with a 1×1 convolution, followed by a 3×3 convolution, and ending with the concatenation of the resulting features. This setup helps the network separately learn spatial features and channel-wise features, which is more efficient than learning them jointly. More-complex versions of an Inception module are also possible, typically involving pooling operations, different spatial convolution sizes (for example, 5×5 instead of 3×3 on some branches), and branches without a spatial convolution (only a 1×1 convolution). An example of such a module, taken from Inception V3, is shown in figure 7.8.





The purpose of 1×1 convolutions

You already know that convolutions extract spatial patches around every tile in an input tensor and apply the same transformation to each patch. An edge case is when the patches extracted consist of a single tile. The convolution operation then becomes equivalent to running each tile vector through a Dense layer: it will compute features that mix together information from the channels of the input tensor, but it won't mix information across space (because it's looking at one tile at a time). Such 1×1 convolutions (also called pointwise convolutions) are featured in Inception modules, where they contribute to factoring out channel-wise feature learning and spacewise feature learning—a reasonable thing to do if you assume that each channel is highly autocorrelated across space, but different channels may not be highly correlated with each other.

Here's how you'd implement the module featured in figure 7.8 using the functional API. This example assumes the existence of a 4D input tensor x :

Every branch has the same stride value (2), which is necessary to keep all branch outputs in this branch, the striding occurs the same size so you can concatenate them.

in the spatial convolution layer.

```
from keras import layers
branch_a = layers.Conv2D(128, 1, activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_a)
branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)
```

```
branch_d = layers.Conv2D(128, 1, activation='relu')(x) branch_d =  
layers.Conv2D(128, 3, activation='relu')(branch_d) branch_d =  
layers.Conv2D(128, 3, activation='relu',  
strides=2)(branch_d)
```

```
output = layers.concatenate( [branch_a, branch_b, branch_c, branch_d],  
axis=-1)
```

Concatenates the branch outputs to

In this branch, the striding occurs obtain the module in the average pooling layer.output

Note that the full Inception V3 architecture is available in Keras as `keras.applications.inception_v3.InceptionV3`, including weights pretrained on the ImageNet dataset. Another closely related model available as part of the Keras applications module is Xception.¹⁶ Xception, which stands for extreme inception, is a convnet architecture loosely inspired by Inception. It takes the idea of separating the learning of channel-wise and space-wise features to its logical extreme, and replaces Inception modules with depthwise separable convolutions consisting of a depthwise convolution (a spatial convolution where every input channel is handled separately) followed by a pointwise convolution (a 1×1 convolution)—effectively, an extreme form of an Inception module, where spatial features and channel-wise features are fully separated. Xception has roughly the same number of parameters as Inception V3, but it shows better runtime performance and higher accuracy on ImageNet as well as other large-scale datasets, due to a more efficient use of model parameters.

RESIDUAL CONNECTIONS

Residual connections are a common graph-like network component found in many post2015 network architectures, including Xception. They were introduced by He et al. from Microsoft in their winning entry in the ILSVRC ImageNet challenge in late 2015.¹⁷ They tackle two common problems that plague any large-scale deep-learning model: vanishing gradients and representational bottlenecks. In general, adding residual connections to any model that has more than 10 layers is likely to be beneficial.

A residual connection consists of making the output of an earlier layer

available as input to a later layer, effectively creating a shortcut in a sequential network. Rather than being concatenated to the later activation, the earlier output is summed with the later activation, which assumes that both activations are the same size. If they're different sizes, you can use a linear transformation to reshape the earlier activation into the target shape (for example, a Dense layer without an activation or, for convolutional feature maps, a 1×1 convolution without an activation).

Here's how to implement a residual connection in Keras when the feature-map sizes are the same, using identity residual connections. This example assumes the existence of a 4D input tensor x :

```
from keras import layers Applies a transformation to  $x$  = ...
```

```
= layers.Conv2D(128, 3, activation='relu', padding='same')(x)y =  
layers.Conv2D(128, 3, activation='relu', padding='same')(y) y =  
layers.Conv2D(128, 3, activation='relu', padding='same')(y) y =  
layers.add([y, x])
```

Adds the original x back to the output features

.

And the following implements a residual connection when the feature-map sizes differ, using a linear residual connection (again, assuming the existence of a 4D input tensor x):

Uses a 1×1 convolution to

```
from keras import layers linearly downsample the original  
 $x$  tensor to the same shape as  $y$   
= ...
```

```
= layers.Conv2D(128, 3, activation='relu', padding='same')(x)y =  
layers.Conv2D(128, 3, activation='relu', padding='same')(y) y =  
layers.MaxPooling2D(2, strides=2)(y) residual = layers.Conv2D(128, 1,  
strides=2, padding='same')(x)
```

```
y = layers.add([y, residual])Adds the residual tensor back to the output
```

features

Representational bottlenecks in deep learning

In a Sequential model, each successive representation layer is built on top of the previous one, which means it only has access to information contained in the activation of the previous layer. If one layer is too small (for example, it has features that are too low-dimensional), then the model will be constrained by how much information can be crammed into the activations of this layer.

(continued)

You can grasp this concept with a signal-processing analogy: if you have an audioprocessing pipeline that consists of a series of operations, each of which takes as input the output of the previous operation, then if one operation crops your signal to a low-frequency range (for example, 0–15 kHz), the operations downstream will never be able to recover the dropped frequencies. Any loss of information is permanent. Residual connections, by reinjecting earlier information downstream, partially solve this issue for deep-learning models.

Vanishing gradients in deep learning

Backpropagation, the master algorithm used to train deep neural networks, works by propagating a feedback signal from the output loss down to earlier layers. If this feedback signal has to be propagated through a deep stack of layers, the signal may become tenuous or even be lost entirely, rendering the network untrainable. This issue is known as vanishing gradients.

This problem occurs both with deep networks and with recurrent networks over very long sequences—in both cases, a feedback signal must be propagated through a long series of operations. You’re already familiar with the solution that the LSTM layer uses to address this problem in recurrent networks: it introduces a carry track that propagates information parallel to the main processing track. Residual connections work in a similar way in feedforward deepnetworks, but they’re even simpler: they introduce a purely linear information carry track parallel to the main layer stack, thus helping to propagate gradients through arbitrarily deep stacks of layers.

Layer weight sharing

One more important feature of the functional API is the ability to reuse a layer instance several times. When you call a layer instance twice, instead of instantiating a new layer for each call, you reuse the same weights with every call. This allows you to build models that have shared branches—several branches that all share the same knowledge and perform the same operations. That is, they share the same representations and learn these representations simultaneously for different sets of inputs.

For example, consider a model that attempts to assess the semantic similarity between two sentences. The model has two inputs (the two sentences to compare) and outputs a score between 0 and 1, where 0 means unrelated sentences and 1 means sentences that are either identical or reformulations of each other. Such a model could be useful in many applications, including deduplicating natural-language queries in a dialog system.

In this setup, the two input sentences are interchangeable, because semantic similarity is a symmetrical relationship: the similarity of A to B is identical to the similarity of B to A. For this reason, it wouldn't make sense to learn two independent models for processing each input sentence. Rather, you want to process both with a single LSTM layer. The representations of this LSTM layer (its weights) are learned based on both inputs simultaneously. This is what we call a Siamese LSTM model or a shared LSTM.

Here's how to implement such a model using layer sharing (layer reuse) in the Keras functional API:

```
Instantiates a single LSTM layer, once
from keras import layers
from keras import Input
```

```
from keras.models import Model Building the left branch of the model:
inputs are variable-length
lstm = layers.LSTM(32)
sequences of vectors of size 128.
```

```
left_input = Input(shape=(None, 128)) left_output = lstm(left_input)Building
```


the right branch of the model:

when you call an existing layer

```
right_input = Input(shape=(None, 128))
```

instance, you reuse its weights.

```
right_output = lstm(right_input)
```

```
merged = layers.concatenate([left_output, right_output], axis=-1) predictions
```

```
= layers.Dense(1, activation='sigmoid')(merged) model = Model([left_input,
```

```
right_input], predictions) model.fit([left_data, right_data], targets)
```

Builds the classifier on top Instantiating and training the model: when you train such a model, the weights of the LSTM layer

are updated based on both inputs.

Naturally, a layer instance may be used more than once—it can be called arbitrarily many times, reusing the same set of weights every time. **Models as layers**

Importantly, in the functional API, models can be used as you'd use layers—effectively, you can think of a model as a “bigger layer.” This is true of both the Sequential and Model classes. This means you can call a model on an input tensor and retrieve an output tensor:

```
y = model(x)
```

If the model has multiple input tensors and multiple output tensors, it should be called with a list of tensors:

```
y1, y2 = model([x1, x2])
```

When you call a model instance, you're reusing the weights of the model—exactly like what happens when you call a layer instance. Calling an instance, whether it's a layer instance or a model instance, will always reuse the existing learned representations of the instance—which is intuitive.

One simple practical example of what you can build by reusing a model instance is a vision model that uses a dual camera as its input: two parallel cameras, a few centimeters (one inch) apart. Such a model can perceive depth, which can be useful in many applications. You shouldn't need two independent models to extract visual features from the left camera and the right camera before merging the two feeds. Such low-level processing can be

shared across the two inputs: that is, done via layers that use the same weights and thus share the same representations. Here's how you'd implement a Siamese vision model (shared convolutional base) in Keras:

```
from keras import layers
```

The base image-processing model is the Xception network

```
from keras import applications
from keras import Input (convolutional base only).
```

```
xception_base = applications.Xception(weights=None, include_top=False)
left_input = Input(shape=(250, 250, 3)) The inputs are 250 × 250 right_input
= Input(shape=(250, 250, 3)) RGB images.
```

```
left_features =
xception_base(left_input) Calls the same vision right_input =
xception_base(right_input) model twice
```

```
merged_features = layers.concatenate( [left_features, right_input], axis=-1)
The merged features contain information from the right visual feed and
the left visual feed.
```

Wrapping up

This concludes our introduction to the Keras functional API—an essential tool for building advanced deep neural network architectures. Now you know the following:

To step out of the Sequential API whenever you need anything more than a linear stack of layers

How to build Keras models with several inputs, several outputs, and complex internal network topology, using the Keras functional API

How to reuse the weights of a layer or model across different processing branches, by calling the same layer or model instance several times

Inspecting and monitoring deep-learning models using Keras callbacks and TensorBoard

In this section, we'll review ways to gain greater access to and control over what goes on inside your model during training. Launching a training run on a large dataset for tens of epochs using `model.fit()` or `model.fit_generator()`

can be a bit like launching a paper airplane: past the initial impulse, you don't have any control over its trajectory or its landing spot. If you want to avoid bad outcomes (and thus wasted paper airplanes), it's smarter to use not a paper plane, but a drone that can sense its environment, send data back to its operator, and automatically make steering decisions based on its current state. The techniques we present here will transform the call to `model.fit()` from a paper airplane into a smart, autonomous drone that can selfintrospect and dynamically take action.

Using callbacks to act on a model during training

When you're training a model, there are many things you can't predict from the start. In particular, you can't tell how many epochs will be needed to get to an optimal validation loss. The examples so far have adopted the strategy of training for enough epochs that you begin overfitting, using the first run to figure out the proper number of epochs to train for, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful.

A much better way to handle this is to stop training when you measure that the validation loss is no longer improving. This can be achieved using a Keras callback. A callback is an object (a class instance implementing specific methods) that is passed to the model in the call to fit and that is called by the model at various points during training. It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

Here are some examples of ways you can use callbacks:

Model checkpointing —Saving the current weights of the model at different points during training.

Early stopping—Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).

Dynamically adjusting the value of certain parameters during training—Such as the learning rate of the optimizer.

Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated—The Keras progress bar that you're familiar with is a callback!

The `keras.callbacks` module includes a number of built-in callbacks (this is not an exhaustive list):

```
keras.callbacks.ModelCheckpoint keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Let's review a few of them to give you an idea of how to use them:

`ModelCheckpoint`, `EarlyStopping`, and `ReduceLROnPlateau`.

THE MODELCHECKPOINT AND EARLYSTOPPING CALLBACKS

You can use the `EarlyStopping` callback to interrupt training once a target metric being monitored has stopped improving for a fixed number of epochs. For instance, this callback allows you to interrupt training as soon as you start overfitting, thus avoiding having to retrain your model for a smaller number of epochs. This callback is typically used in combination with `ModelCheckpoint`, which lets you continually save the model during training (and, optionally, save only the current best model so far: the version of the model that achieved the best performance at the end of an epoch):

Callbacks are passed to the model via the `callbacks` argument in `fit`, which takes a list of callbacks. You can pass any number of callbacks.

Monitors the model's

```
import keras validation accuracy
```

```
callbacks_list = [ Interrupts training when keras.callbacks.EarlyStopping(accuracy has stopped monitor='acc', improving for more than one
patience=1, epoch (that is, two epochs)
```

```
),
```

```
keras.callbacks.ModelCheckpoint(every epoch filepath='my_model.h5',
monitor='val_loss', save_best_only=True, won't overwrite the
```

```
Saves the current weights after Path to the destination model file These
```

two arguments mean you

) **model file unless val_loss has improved, which allows] you to keep the best model seen during training.**

```
model.compile(optimizer='rmsprop',  
loss='binary_crossentropy', You monitor
```

```
metrics=['acc']) be part of the model's
```

```
model.fit(x, y,
```

Note that because the callback will
validation accuracy, you need to pass

```
epochs=10,
```

```
batch_size=32,
```

```
callbacks=callbacks_list,
```

```
validation_data to the call to fit.
```

```
validation_data=(x_val, y_val))
```

THE REDUCELRONPLATEAU CALLBACK

accuracy, so it should metrics.

monitor validation loss and

You can use this callback to reduce the learning rate when the validation loss has stopped improving. Reducing or increasing the learning rate in case of a loss plateau is an effective strategy to get out of local minima during training. The following example uses the ReduceLROnPlateau callback:

```
callbacks_list = [Monitors the model's
```

```
keras.callbacks.ReduceLROnPlateau(validation loss monitor='val_loss'
```

```
factor=0.1, Divides the learning rate by 10 when triggered patience=10,
```

) **The callback is triggered after the validation**

loss has stopped improving for 10 epochs.

```
] model.fit(x, y,
```

```
epochs=10, Because the callback will
```

```
batch_size=32, monitor the validation loss, you
```

```
callbacks=callbacks_list, need to pass validation_data to validation_data=
```

(x_val, y_val))**the call to fit.**

WRITING YOUR OWN CALLBACK

If you need to take a specific action during training that isn't covered by one of the built-in callbacks, you can write your own callback. Callbacks are implemented by subclassing the class `keras.callbacks.Callback`. You can then implement any number of the following transparently named methods, which are called at various points during training:

`on_epoch_begin`**Called at the start of every epoch** `on_epoch_end`**Called at the end of every epoch**

`on_batch_begin` **Called right before processing each batch**
`on_batch_end`**Called right after processing each batch**

`on_train_begin`**Called at the start of training** `on_train_end`**Called at the end of training**

These methods all are called with a `logs` argument, which is a dictionary containing information about the previous batch, epoch, or training run: training and validation metrics, and so on. Additionally, the callback has access to the following attributes:

`self.model` —The model instance from which the callback is being called
`self.validation_data`—The value of what was passed to `fit` as validation data

Here's a simple example of a custom callback that saves to disk (as Numpy arrays) the activations of every layer of the model at the end of every epoch, computed on the first sample of the validation set:

```
import keras
import numpy as np

class ActivationLogger(keras.callbacks.Callback):
    def on_epoch_end(self, logs=None):
```

before training, to inform the callback of what model

```
    def set_model(self, model):
        self.model = model
        self.layer_outputs = [layer.output for layer in model.layers]
```

```
self.activations_model = keras.models.Model(model.input,  
layer_outputs)
```

```
def on_epoch_end(self, epoch, logs=None): Model instance if  
self.validation_data is None: that returns the raise RuntimeError('Requires  
validation_data.') activations of every layer
```

```
validation_sample = self.validation_data[0][0:1] activations =  
self.activations_model.predict(validation_sample) f =  
open('activations_at_epoch_' + str(epoch) + '.npz', 'w') np.savez(f,  
activations) f.close()
```

**Obtains the first input sample Saves
arrays to disk
of the validation data**

This is all you need to know about callbacks —the rest is technical details, which you can easily look up. Now you're equipped to perform any sort of logging or preprogrammed intervention on a Keras model during training.

Introduction to TensorBoard:
the TensorFlow visualization framework

To do good research or develop good models, you need rich, frequent feedback about what's going on inside your models during your experiments. That's the point of running experiments: to get information about how well a model performs—as much information as possible. Making progress is an iterative process, or loop: you start with an idea and express it as an experiment, attempting to validate or invalidate your idea. You run this experiment and process the information it generates. This inspires your next idea. The more iterations of this loop you're able to run, the more refined and powerful your ideas become. Keras helps you go from idea to experiment in the least possible time, and fast GPUs can help you get from experiment to result as quickly as possible. But what about processing the experiment results? That's where TensorBoard comes in.

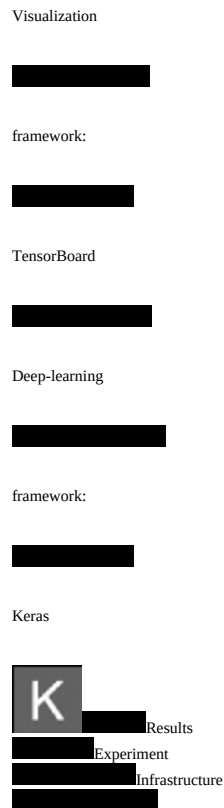


Figure 7.9 The loop of progress

This section introduces TensorBoard, a browser-based visualization tool that comes packaged with TensorFlow. Note that it's only available for Keras models when you're using Keras with the TensorFlow backend.

The key purpose of TensorBoard is to help you visually monitor everything that goes on inside your model during training. If you're monitoring more information than just the model's final loss, you can develop a clearer vision of what the model does and doesn't do, and you can make progress more quickly. TensorBoard gives you access to several neat features, all in your browser:

- Visually monitoring metrics during training

- Visualizing your model architecture

- Visualizing histograms of activations and gradients

- Exploring embeddings in 3D

Let's demonstrate these features on a simple example. You'll train a 1D convnet on the IMDB sentiment-analysis task.

The model is similar to the one you saw in the last section of chapter 6. You'll consider only the top 2,000 words in the IMDB vocabulary, to make visualizing word embeddings more tractable.

Text-classification model to use with TensorBoard

```
import keras
```

```
from keras import  
from keras.datasets  
features from  
sequence
```

Cuts off texts after this number layers **Number of words to** import imdb
consider as keras.preprocessing import

```
max_features = 2000 of words (among max_features  
most common words)  
max_len = 500
```

```
(x_train, y_train), (x_test, y_test) =  
imdb.load_data(num_words=max_features) x_train =  
sequence.pad_sequences(x_train, maxlen=max_len) x_test =  
sequence.pad_sequences(x_test, maxlen=max_len)
```

```
model = keras.models.Sequential()  
model.add(layers.Embedding(max_features, 128, input_length=max_len,  
name='embed'))
```

```
model.add(layers.Conv1D(32, 7, activation='relu'))  
model.add(layers.MaxPooling1D(5)) model.add(layers.Conv1D(32, 7,  
activation='relu')) model.add(layers.GlobalMaxPooling1D())  
model.add(layers.Dense(1)) model.summary()  
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=  
['acc'])
```

Before you start using TensorBoard, you need to create a directory where you'll store the log files it generates.

Creating a directory for TensorBoard log files

```
$ mkdir my_log_dir
```

Let's launch the training with a TensorBoard callback instance. This callback will write log events to disk at the specified location.

Inspecting and monitoring deep-learning models using Keras callbacks and TensorBoard 454

Training the model with a **TensorBoard** callback `callbacks = [`
`keras.callbacks.TensorBoard(`

```
log_dir='my_log_dir',  
histogram_freq=1,  
embeddings_freq=1,
```

```
) ]
```

```
history = model.fit(x_train,  
y_train,  
epochs=20, batch_size=128,
```

Log files will be written at this location.

Records activation histograms every 1 epoch Records embedding data every 1 epoch

```
validation_split=0.2, callbacks=callbacks)
```

At this point, you can launch the TensorBoard server from the command line, instructing it to read the logs the callback is currently writing. The `tensorboard` utility should have been automatically installed on your machine the moment you installed TensorFlow (for example, via `pip`):

```
$ tensorboard --logdir=my_log_dir
```

You can then browse to `http://localhost:6006` and look at your model training (see figure 7.10). In addition to live graphs of the training and validation metrics, you get access to the Histograms tab, where you can find pretty visualizations of histograms of activation values taken by your layers (see figure 7.11).

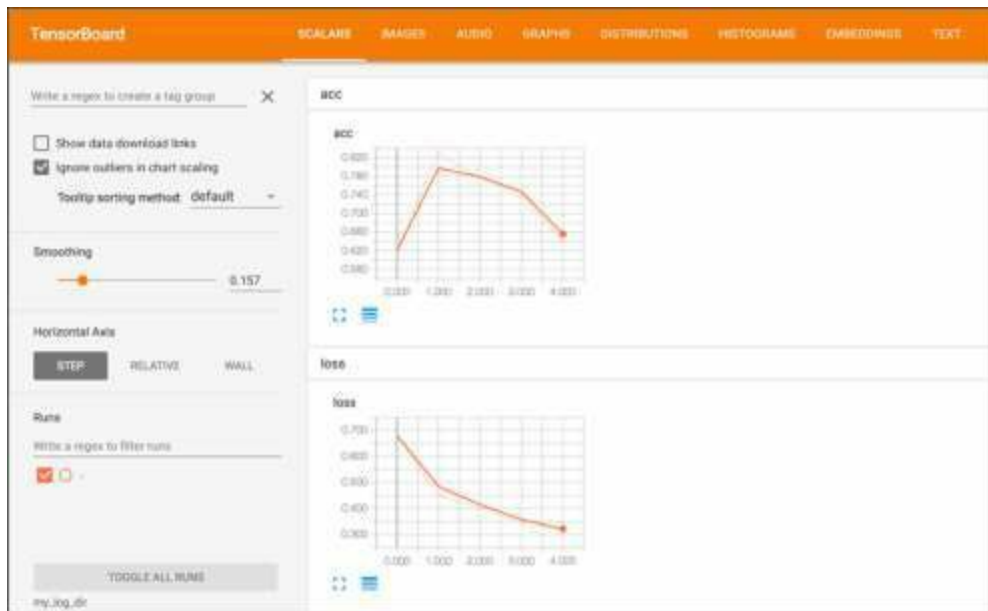


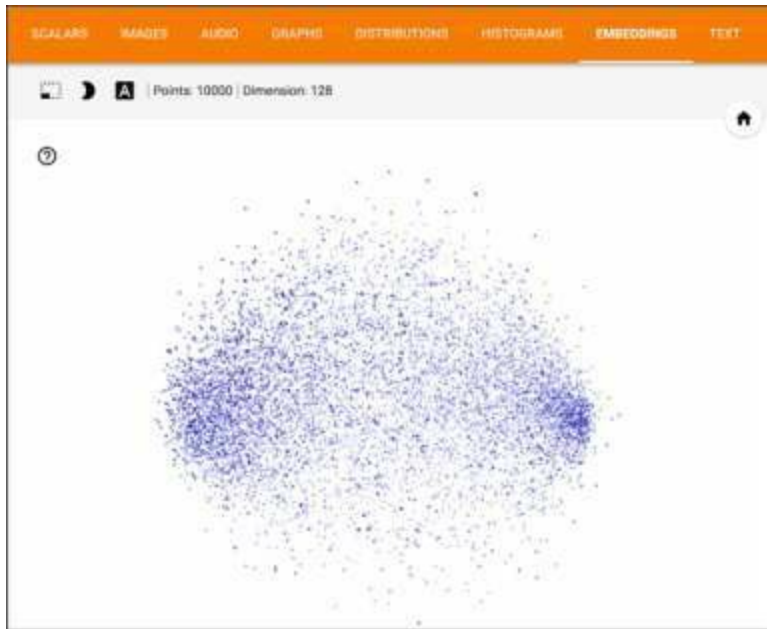
Figure 7.10 TensorBoard: metrics monitoring



Figure 7.11 TensorBoard: activation histograms

The Embeddings tab gives you a way to inspect the embedding locations and spatial relationships of the 10,000 words in the input vocabulary, as learned by dimensional, TensorBoard automatically reduces it to 2D or 3D using a dimensionality-reduction algorithm of your choice: either principal component analysis (PCA) or t-distributed stochastic neighbor embedding (t-SNE). In figure 7.12, in the point cloud, you can clearly see two clusters:

words with a positive connotation and words with a negative connotation. The visualization makes it immediately obvious that embeddings trained jointly with a specific objective result in models that are completely specific to the underlying task—that’s the reason using pretrained generic word embeddings is rarely a good idea.



TensorBoard: interactive 3D word-embedding visualization

The Graphs tab shows an interactive visualization of the graph of low-level TensorFlow operations underlying your Keras model (see figure 7.13). As you can see, there’s a lot more going on than you would expect. The model you just built may look simple when defined in Keras—a small stack of basic layers—but under the hood, you need to construct a fairly complex graph structure to make it work. A lot of it is related to the gradient-descent process. This complexity differential between what you see and what you’re manipulating is the key motivation for using Keras as your way of building models, instead of working with raw TensorFlow to define everything from scratch.

Keras makes your workflow dramatically simpler.

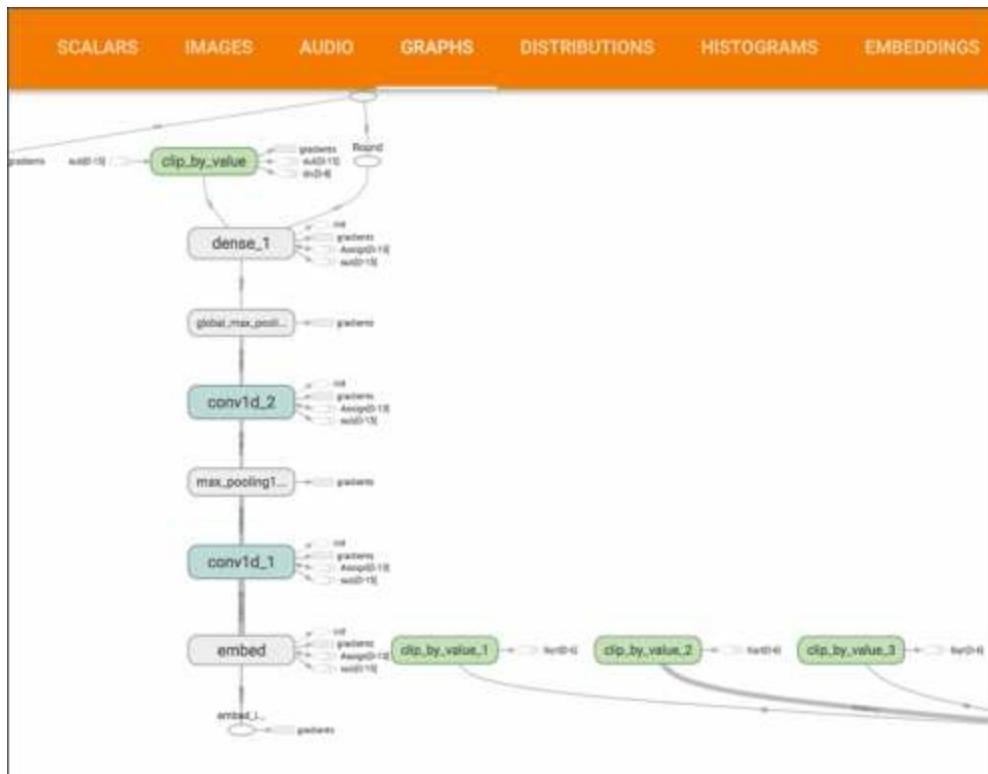


Figure 7.13 TensorBoard: TensorFlow graph visualization

Note that Keras also provides another, cleaner way to plot models as graphs of layers rather than graphs of TensorFlow operations: the utility `keras.utils.plot_model`. Using it requires that you've installed the Python `pydot` and `pydot-ng` libraries as well as the `graphviz` library. Let's take a quick look:

```
from keras.utils import plot_model plot_model(model, to_file='model.png')
```

This creates the PNG image shown in

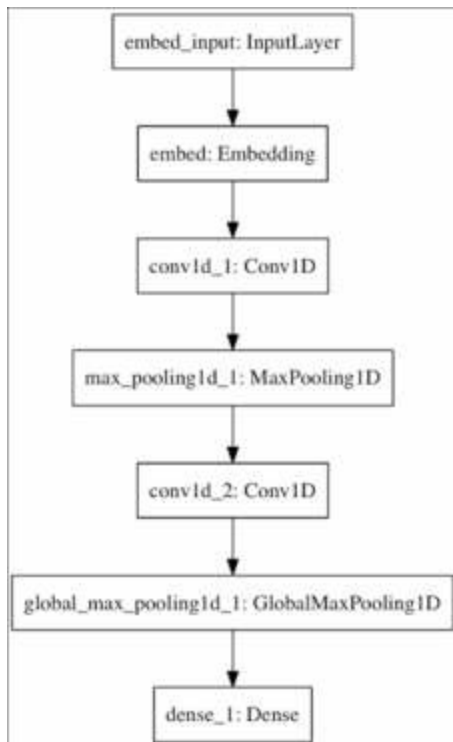
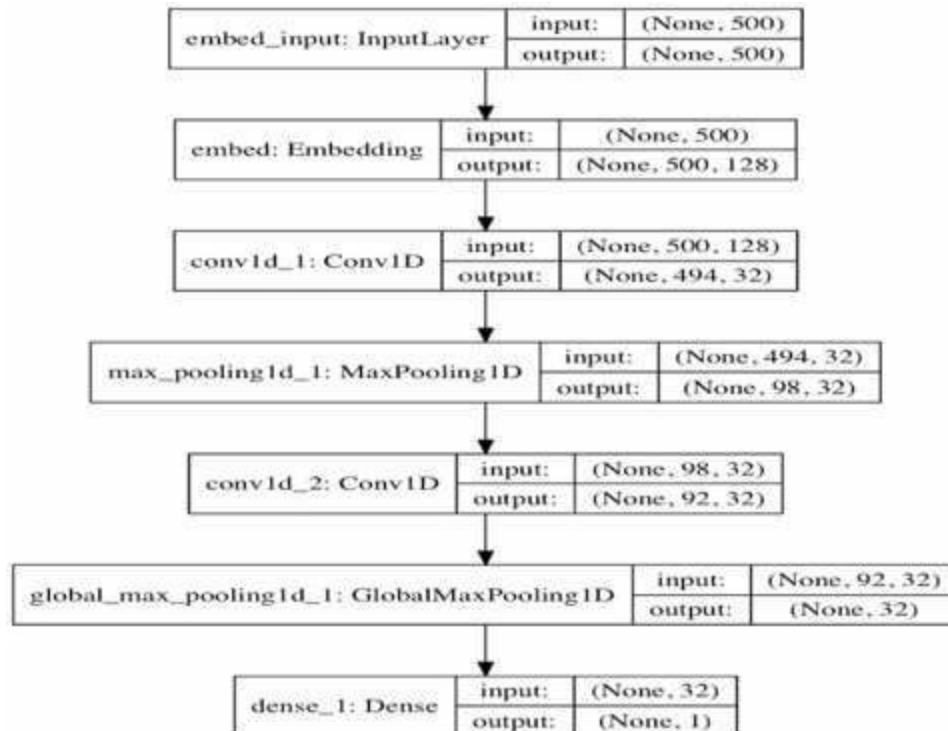


Figure 7.14 A model plot as a graph of layers, generated with **plot_model**

You also have the option of displaying shape information in the graph of layers. This example visualizes model topology using `plot_model` and the `show_shapes` option (see

```
from keras.utils import plot_model plot_model(model, show_shapes=True,  
to_file='model.png')
```



A model plot with shape information

Wrapping up

Keras callbacks provide a simple way to monitor models during training and automatically take action based on the state of the model.

When you're using TensorFlow, TensorBoard is a great way to visualize model activity in your browser. You can use it in Keras models via the TensorBoard callback.

Getting the most out of your models

Trying out architectures blindly works well enough if you just need something that works okay. In this section, we'll go beyond "works okay" to "works great and wins machinelearning competitions" by offering you a quick guide to a set of must-know techniques for building state-of-the-art deep-learning models.

Advanced architecture patterns

We covered one important design pattern in detail in the previous section: residual connections. There are two more design patterns you should know about: normalization and depthwise separable convolution. These patterns are especially relevant when you're building high-performing deep convnets, but

they're commonly found in many other types of architectures as well.

BATCH NORMALIZATION

Normalization is a broad category of methods that seek to make different samples seen by a machine-learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is one you've seen several times in this book already: centering the data on 0 by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit variance:

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Previous examples normalized data before feeding it into models. But data normalization should be a concern after every transformation operated by the network: even if the data entering a Dense or Conv2D network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming out.

Batch normalization is a type of layer (BatchNormalization in Keras) introduced in 2015 by Ioffe and Szegedy;¹⁸ it can adaptively normalize data even as the mean and variance change over time during training. It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training. The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks. Some very deep networks can only be trained if they include multiple BatchNormalization layers. For instance, BatchNormalization is used liberally in many of the advanced convnet architectures that come packaged with Keras, such as ResNet50, Inception V3, and Xception.

The BatchNormalization layer is typically used after a convolutional or densely connected layer: `conv_model.add(layers.Conv2D(32, 3, activation='relu'))` **After a Conv layer**
`conv_model.add(layers.BatchNormalization())`


```
dense_model.add(layers.Dense(32, activation='relu')) After a Dense layer  
dense_model.add(layers.BatchNormalization())
```

The BatchNormalization layer takes an axis argument, which specifies the feature axis that should be normalized. This argument defaults to -1, the last axis in the input tensor. This is the correct value when using Dense layers, Conv1D layers, RNN layers, and Conv2D layers with data_format set to "channels_last". But in the niche use case of Conv2D layers with data_format set to "channels_first", the features axis is axis 1; the axis argument in BatchNormalization should accordingly be set to 1.

Batch renormalization

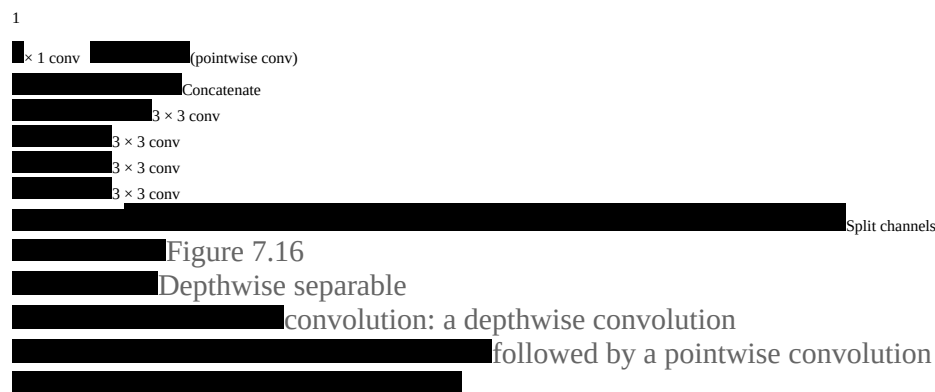
A recent improvement over regular batch normalization is batch renormalization, introduced by Ioffe in 2017.^a It offers clear benefits over batch normalization, at no apparent cost. At the time of writing, it's too early to tell whether it will supplant batch normalization—but I think it's likely. Even more recently, Klambauer et al. introduced self-normalizing neural networks,^b which manage to keep data normalized after going through any Dense layer by using a specific activation function (selu) and a specific initializer (lecun_normal). This scheme, although highly interesting, is limited to densely connected networks for now, and its usefulness hasn't yet been broadly replicated.

^a Sergey Ioffe, "Batch Renormalization: Towards Reducing Minibatch Dependence in BatchNormalized Models" (2017), <https://arxiv.org/abs/1702.03275>. ^bGünter Klambauer et al., "Self-Normalizing Neural Networks," Conference on Neural Information Processing Systems (2017), <https://arxiv.org/abs/1706.02515>.

DEPTHWISE SEPARABLE CONVOLUTION

What if I told you that there's a layer you can use as a drop-in replacement for Conv2D that will make your model lighter (fewer trainable weight parameters) and faster (fewer floating-point operations) and cause it to perform a few percentage points better on its task? That is precisely what the depthwise separable convolution layer does (SeparableConv2D). This layer

performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution (a 1×1 convolution), as shown in figure 7.16. This is equivalent to separating the learning of spatial features and the learning of channel-wise features, which makes a lot of sense if you assume that spatial locations in the input are highly correlated, but different channels are fairly independent. It requires significantly fewer parameters and involves fewer computations, thus resulting in smaller, speedier models. And because it's a more representationally efficient way to perform convolution, it tends to learn better representations using less data, resulting in better-performing models.



These advantages become especially important when you're training small models from scratch on limited data. For instance, here's how you can build a lightweight, depthwise separable convnet for an image-classification task (softmax categorical classification) on a small dataset:

```
from keras.models import Sequential, Model
from keras import layers

height = 64
width = 64
channels = 3
num_classes = 10

model = Sequential()
model.add(layers.SeparableConv2D(32, 3,
    activation='relu', input_shape=(height, width, channels)))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))
```

```
model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.GlobalAveragePooling2D())
```

```
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy')
```

When it comes to larger-scale models, depthwise separable convolutions are the basis of the Xception architecture, a high-performing convnet that comes packaged with Keras. You can read more about the theoretical grounding for depthwise separable convolutions and Xception in my paper “Xception: Deep Learning with Depthwise Separable Convolutions.”¹⁹

Hyperparameter optimization

When building a deep-learning model, you have to make many seemingly arbitrary decisions: How many layers should you stack? How many units or filters should go in each layer? Should you use relu as activation, or a different function? Should you use BatchNormalization after a given layer? How much dropout should you use? And so on. These architecture-level parameters are called hyperparameters to distinguish them from the parameters of a model, which are trained via backpropagation.

In practice, experienced machine-learning engineers and researchers build intuition over time as to what works and what doesn’t when it comes to these choices— they develop hyperparameter-tuning skills. But there are no formal rules. If you want to get to the very limit of what can be achieved on a given task, you can’t be content with arbitrary choices made by a fallible human. Your initial decisions are almost always suboptimal, even if you have good intuition. You can refine your choices by tweaking them by hand and retraining the model repeatedly—that’s what machinelearning engineers and researchers spend most of their time doing. But it shouldn’t be your job as a human to fiddle with hyperparameters all day—that is better left to a machine.

Thus you need to explore the space of possible decisions automatically, systematically, in a principled way. You need to search the architecture space

and find the bestperforming ones empirically. That's what the field of automatic hyperparameter optimization is about: it's an entire field of research, and an important one. The process of optimizing hyperparameters typically looks like this:

Choose a set of hyperparameters (automatically).

Build the corresponding model.

Fit it to your training data, and measure the final performance on the validation data. Choose the next set of hyperparameters to try (automatically).

Repeat.

Eventually, measure performance on your test data.

The key to this process is the algorithm that uses this history of validation performance, given various sets of hyperparameters, to choose the next set of hyperparameters to evaluate. Many different techniques are possible: Bayesian optimization, genetic algorithms, simple random search, and so on.

¹⁹ See note 5 above.

Training the weights of a model is relatively easy: you compute a loss function on a mini-batch of data and then use the Backpropagation algorithm to move the weights in the right direction. Updating hyperparameters, on the other hand, is extremely challenging. Consider the following:

Computing the feedback signal (does this set of hyperparameters lead to a highperforming model on this task?) can be extremely expensive: it requires creating and training a new model from scratch on your dataset.

The hyperparameter space is typically made of discrete decisions and thus isn't continuous or differentiable. Hence, you typically can't do gradient descent in hyperparameter space. Instead, you must rely on gradient-free optimization techniques, which naturally are far less efficient than gradient descent.

Because these challenges are difficult and the field is still young, we currently only have access to very limited tools to optimize models. Often, it turns out that random search (choosing hyperparameters to evaluate at random, repeatedly) is the best solution, despite being the most naive one.

But one tool I have found reliably better than random search is Hyperopt (<https://github.com/hyperopt/hyperopt>), a Python library for hyperparameter optimization that internally uses trees of Parzen estimators to predict sets of hyperparameters that are likely to work well. Another library called Hyperas (<https://github.com/maxpumperla/hyperas>) integrates Hyperopt for use with Keras models. Do check it out.

NOTE One important issue to keep in mind when doing automatic hyperparameter optimization at scale is validationset overfitting. Because you're updating hyperparameters based on a signal that is computed using your validation data, you're effectively training them on the validation data, and thus they will quickly overfit to the validation data. Always keep this in mind.

Overall, hyperparameter optimization is a powerful technique that is an absolute requirement to get to state-of-the-art models on any task or to win machine-learning competitions. Think about it: once upon a time, people handcrafted the features that went into shallow machine-learning models. That was very much suboptimal. Now, deep learning automates the task of hierarchical feature engineering—features are learned using a feedback signal, not handtuned, and that's the way it should be. In the same way, you shouldn't handcraft your model architectures; you should optimize them in a principled way. At the time of writing, the field of automatic hyperparameter optimization is very young and immature, as deep learning was some years ago, but I expect it to boom in the next few years.

Model ensembling

Another powerful technique for obtaining the best possible results on a task is model ensembling. Ensembling consists of pooling together the predictions of a set of different models, to produce better predictions. If you look at machine-learning competitions, in particular on Kaggle, you'll see that the winners use very large ensembles of models that inevitably beat any single model, no matter how good.

Ensembling relies on the assumption that different good models trained independently are likely to be good for different reasons: each model looks at slightly different aspects of the data to make its predictions, getting part of the “truth” but not all of it. You may be familiar with the ancient parable of

the blind men and the elephant: a group of blind men come across an elephant for the first time and try to understand what the elephant is by touching it. Each man touches a different part of the elephant's body—just one part, such as the trunk or a leg. Then the men describe to each other what an elephant is: “It's like a snake,” “Like a pillar or a tree,” and so on. The blind men are essentially machine-learning models trying to understand the manifold of the training data, each from its own perspective, using its own assumptions (provided by the unique architecture of the model and the unique random weight initialization). Each of them gets part of the truth of the data, but not the whole truth. By pooling their perspectives together, you can get a far more accurate description of the data. The elephant is a combination of parts: not any single blind man gets it quite right, but, interviewed together, they can tell a fairly accurate story.

Let's use classification as an example. The easiest way to pool the predictions of a set of classifiers (to ensemble the classifiers) is to average their predictions at inference time:

Use four different models to compute initial predictions.

```
preds_a = model_a.predict(x_val)
```

```
preds_b = model_b.predict(x_val) This new prediction array preds_c =  
model_c.predict(x_val) should be more accurate preds_d =  
model_d.predict(x_val) than any of the initial ones.
```

```
final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d)
```

This will work only if the classifiers are more or less equally good. If one of them is significantly worse than the others, the final predictions may not be as good as the best classifier of the group.

A smarter way to ensemble classifiers is to do a weighted average, where the weights are learned on the validation data—typically, the better classifiers are given a higher weight, and the worse classifiers are given a lower weight. To search for a good set of ensembling weights, you can use random search or a simple optimization algorithm such as Nelder-Mead:

```
preds_a =  
model_a.predict(x_val)
```

```
preds_b = model_b.predict(x_val) These weights (0.5, 0.25, preds_c =
```

`model_c.predict(x_val)` **0.1, 0.15) are assumed to** `preds_d =`
`model_d.predict(x_val)` **be learned empirically.** `final_preds = 0.5 * preds_a`
`+ 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d`

There are many possible variants: you can do an average of an exponential of the predictions, for instance. In general, a simple weighted average with weights optimized on the validation data provides a very strong baseline.

The key to making ensembling work is the diversity of the set of classifiers. Diversity is strength. If all the blind men only touched the elephant's trunk, they would agree that elephants are like snakes, and they would forever stay ignorant of the truth of the elephant. Diversity is what makes ensembling work. In machine-learning terms, if all of your models are biased in the same way, then your ensemble will retain this same bias. If your models are biased in different ways, the biases will cancel each other out, and the ensemble will be more robust and more accurate.

For this reason, you should ensemble models that are as good as possible while being as different as possible. This typically means using very different architectures or even different brands of machine-learning approaches. One thing that is largely not worth doing is ensembling the same network trained several times independently, from different random initializations. If the only difference between your models is their random initialization and the order in which they were exposed to the training data, then your ensemble will be low-diversity and will provide only a tiny improvement over any single model.

One thing I have found to work well in practice—but that doesn't generalize to every problem domain—is the use of an ensemble of tree-based methods (such as random forests or gradient-boosted trees) and deep neural networks. In 2014, partner Andrei Kolev and I took fourth place in the Higgs Boson decay detection challenge on Kaggle (www.kaggle.com/c/higgs-boson) using an ensemble of various tree models and deep neural networks. Remarkably, one of the models in the ensemble originated from a different method than the others (it was a regularized greedy forest) and had a significantly worse score than the others. Unsurprisingly, it was assigned a small weight in the ensemble. But to our surprise, it turned out to improve the overall ensemble by a large factor, because it was so different from every other model: it

provided information that the other models didn't have access to. That's precisely the point of ensembling. It's not so much about how good your best model is; it's about the diversity of your set of candidate models.

In recent times, one style of basic ensemble that has been very successful in practice is the wide and deep category of models, blending deep learning with shallow learning. Such models consist of jointly training a deep neural network with a large linear model. The joint training of a family of diverse models is yet another option to achieve model ensembling.

Wrapping up

When building highperforming deep convnets, you'll need to use residual connections, batch normalization, and depthwise separable convolutions. In the future, it's likely that depthwise separable convolutions will completely replace regular convolutions, whether for 1D, 2D, or 3D applications, due to their higher representational efficiency.

Building deep networks requires making many small hyperparameter and architecture choices, which together define how good your model will be. Rather than basing these choices on intuition or random chance, it's better to systematically search hyperparameter space to find optimal choices. At this time, the process is expensive, and the tools to do it aren't very good. But the Hyperopt and Hyperas libraries may be able to help you. When doing hyperparameter optimization, be mindful of validation-set overfitting!

Winning machine-learning competitions or otherwise obtaining the best possible results on a task can only be done with large ensembles of models. Ensembling via a welloptimized weighted average is usually good enough. Remember: diversity is strength. It's largely pointless to ensemble very similar models; the best ensembles are sets of models that are as dissimilar as possible (while having as much predictive power as possible, naturally).

6 FIXED REGRESSIONS

This is the list of changes to between each release. For full details, see the commit logs. For install and upgrade instructions, see *Installation*. These are the changes in 1.2.3. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in *to_excel()* raising *KeyError* when giving duplicate columns with columns attribute (GH39695)

Fixed regression in nullable integer unary ops propagating mask on assignment (GH39943)

Fixed regression in *DataFrame.__setitem__()* not aligning *DataFrame* on right-hand side for boolean indexer (GH39931)

Fixed regression in *to_json()* failing to use compression with URL-like paths that are internally opened in binary mode or with user-provided file objects that are opened in binary mode (GH39985)

Fixed regression in *Series.sort_index()* and *DataFrame.sort_index()*, which exited with an ungraceful error when having kwarg *ascending=None* passed. Passing *ascending=None* is still considered invalid, and the improved error message suggests a proper usage (*ascending* must be a boolean or a list-like of boolean) (GH39434)

Fixed regression in *DataFrame.transform()* and *Series.transform()* giving incorrect column labels when passed a dictionary with a mix of list and non-list values (GH40018) **Contributors**

A total of 14 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Daniel Saxton

Dylan Percy +

Jody Klymak +

Maxim Ivanov

MeeseeksMachine

Development Team

Richard Shadrach

Rohit Gupta +

Simon Hawkins

Torsten Wörtwein

jbrockmendel

morrme +

partev

patrick

What's new in 1.2.2

These are the changes in 1.2.2. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in *read_excel()* that caused it to raise `AttributeError` when checking version of older `xlrd` versions (GH38955)

Fixed regression in *DataFrame* constructor reordering element when construction from `datetime ndarray` with `dtype` not `"datetime64[ns]"` (GH39422)

Fixed regression in *DataFrame.astype()* and *Series.astype()* not casting to `bytes dtype` (GH39474)

Fixed regression in *to_pickle()* failing to create `bz2/xz` compressed pickle files with `protocol=5` (GH39002)

Fixed regression in *.testing.assert_series_equal()* and *.testing.assert_frame_equal()* always raising `AssertionError` when comparing extension dtypes (GH39410) Fixed regression in *to_csv()* opening `codecs.StreamWriter` in `binary` mode instead of in `text` mode and ignoring user-provided mode (GH39247)

Fixed regression in *Categorical.astype()* casting to incorrect `dtype` when `np.int32` is passed to `dtype` argument (GH39402)

Fixed regression in *to_excel()* creating corrupt files when appending (`mode="a"`) to an existing file (GH39576)

Fixed regression in *DataFrame.transform()* failing in case of an empty *DataFrame* or *Series* (GH39636)

Fixed regression in *groupby()* or *resample()* when aggregating an all-`NaN` or numeric object `dtype` column (GH39329)

Fixed regression in *Rolling.count()* where the `min_periods` argument would be set to 0 after the operation (GH39554)

Fixed regression in *read_excel()* that incorrectly raised when the argument `io` was a non-path and `nonbuffer` and the `engine` argument was specified (GH39528) **Bug fixes**

.read_excel() error message when a specified sheetname does not exist is now uniform across engines (GH39250)

Fixed bug in *.read_excel()* producing incorrect results when the engine `openpyxl` is used and the excel file is missing or has incorrect dimension information; the fix requires `openpyxl >= 3.0.0`, prior versions may still fail (GH38956, GH39001)

Fixed bug in *.read_excel()* sometimes producing a `DataFrame` with trailing rows of `np.nan` when the engine `openpyxl` is used (GH39181)

What's new in

in *.testing.assert_frame_equal()* raising `TypeError` with `check_like=True` when `Index` or `columns` have mixed `dtype` (GH39168)

We have reverted a commit that resulted in several plotting related regressions in 1.2.0 (GH38969, GH38736, GH38865, GH38947 and GH39126). As a result, bugs reported as fixed in 1.2.0 related to inconsistent tick labeling in bar plots are again present (GH26186 and GH11465)

Calling NumPy ufuncs on non-aligned DataFrames

Before 1.2.0, calling a NumPy ufunc on non-aligned `DataFrames` (or `DataFrame` / `Series` combination) would ignore the indices, only match the inputs by shape, and use the index/columns of the first `DataFrame` for the result:

```
>>> df1 = pd.DataFrame({"a": [1, 2], "b": [3, 4]}, index=[0, 1]) ...
df2 = pd.DataFrame({"a": [1, 2], "b": [3, 4]}, index=[1, 2])

>>> df1 a b 0 1 3
      2 4
>>> df2 a b 1 1 3
```

```
2 4
>>> np.add(df1, df2) a b 0 2 6
1 4 8
```

This contrasts with how other operations work, which first align the inputs:

```
>>> df1 + df2 a b
NaN NaN
3.0 7.0
```

, we refactored how NumPy ufuncs are called on DataFrames, and this started to align the inputs first (GH39184), as happens in other operations and as it happens for ufuncs called on Series objects.

For 1.2.1, we restored the previous behaviour to avoid a breaking change, but the above example of `np.add(df1, df2)` with non-aligned inputs will now to raise a warning, and a future 2.0 release will start aligning the inputs first (GH39184). Calling a NumPy ufunc on Series objects (eg `np.add(s1, s2)`) already aligns and continues to do so.

To avoid the warning and keep the current behaviour of ignoring the indices, convert one of the arguments to a NumPy array:

```
>>> np.add(df1, np.asarray(df2)) a b 0 2 6
1 4 8
```

To obtain the future behaviour and silence the warning, you can align manually before passing the arguments to the ufunc:

```
>>> df1, df2 = df1.align(df2)
>>> np.add(df1, df2) a b
NaN NaN
3.0 7.0
NaN NaN
```

Bug fixes

Bug in `read_csv()` with `float_precision="high"` caused segfault or wrong parsing of long exponent strings. This resulted in a regression in some cases as the default for `float_precision` was changed in 1.2.0 (GH38753)

Bug in `read_csv()` not closing an opened file handle when a `csv.Error` or `UnicodeDecodeError` occurred while initializing (GH39024)

Bug in `.testing.assert_index_equal()` raising `TypeError` with `check_order=False` when `Index` has mixed dtype (GH39168)

Other

The deprecated attributes `_AXIS_NAMES` and `_AXIS_NUMBERS` of `DataFrame` and `Series` will no longer show up in `dir` or `inspect.getmembers` calls (GH38740) Bumped minimum fastparquet version to 0.4.0 to avoid `AttributeError` from numba (GH38344)

Bumped minimum pymysql version to 0.8.1 to avoid test failures (GH38344)
Fixed build failure on MacOS 11 in Python 3.9.1 (GH38766)

Added reference to backwards incompatible `check_freq` arg of `testing.assert_frame_equal()` and `testing.assert_series_equal()` in 1.1.0 *whats new* (GH34050)

kylekeppler

mzeitlin11

patrick

5.1.4 What's new in 1.2.0 (December 26, 2020)

These are the changes in 1.2.0. See *Release notes* for a full changelog including other versions of .

Warning: The `xlwt` package for writing old-style `.xls` excel files is no longer maintained. The `xlrd` package is now only for reading old-style `.xls` files.

Previously, the default argument `engine=None` to `read_excel()` would result in using the `xlrd` engine in many cases, including new Excel 2007+ (`.xlsx`) files. If `openpyxl` is installed, many of these cases will now default to using the `openpyxl` engine. See the `read_excel()` documentation for more details.

Thus, it is strongly encouraged to install `openpyxl` to read Excel 2007+ (`.xlsx`) files. Please do not report issues when using ```xlrd``` to read ```.xlsx``` files. This is no longer supported, switch to using `openpyxl` instead.

Attempting to use the `xlwt` engine will raise a `FutureWarning` unless the option `io.excel.xls.writer` is set to `"xlwt"`. While this option is now deprecated

and will also raise a FutureWarning, it can be globally set and the warning suppressed. Users are recommended to write .xlsx files using the openpyxl engine instead.

Enhancements

Optionally disallow duplicate labels

Series and *DataFrame* can now be created with `allows_duplicate_labels=False` flag to control whether the index or columns can contain duplicate labels (GH28394). This can be used to prevent accidental introduction of duplicate labels, which can affect downstream operations.

By default, duplicates continue to be allowed.

In [1]: `pd.Series([1, 2], index=['a', 'a'])` **Out[1]:**

a 1 a 2

Length: 2, dtype: int64

In [2]: `pd.Series([1, 2], index=['a',`

`'a']).set_flags(allows_duplicate_labels=False) ... DuplicateLabelError: Index has duplicates.`

positions

label

a [0, 1]

will propagate the `allows_duplicate_labels` property through many operations. **In [3]:** `a = (`

`...: pd.Series([1, 2], index=['a', 'b'])`

`...: .set_flags(allows_duplicate_labels=False) ...: )` **In [4]:** `a`

(continues on next page)

(continued from previous page)

Out[4]:

a 1 b 2

Length: 2, dtype: int64

An operation introducing duplicates

In [5]: `a.reindex(['a', 'b', 'a']) ...`

`DuplicateLabelError: Index has duplicates.`

positions

```
label
a [0, 2]
[1 rows x 1 columns]
```

Warning: This is an experimental feature. Currently, many methods fail to propagate the `allows_duplicate_labels` value. In future versions it is expected that every method taking or returning one or more `DataFrame` or `Series` objects will propagate `allows_duplicate_labels`.

See *Duplicate Labels* for more.

The `allows_duplicate_labels` flag is stored in the new `DataFrame.flags` attribute. This stores global attributes that apply to the *object*. This differs from `DataFrame.attrs`, which stores information that applies to the dataset.

Passing arguments to fsspec backends

Many read/write functions have acquired the `storage_options` optional argument, to pass a dictionary of parameters to the storage backend. This allows, for example, for passing credentials to S3 and GCS storage. The details of what parameters can be passed to which backends can be found in the documentation of the individual storage backends (detailed from the fsspec docs for builtin implementations and linked to external ones). See Section *Reading/writing remote files*.

GH35655 added fsspec support (including `storage_options`) for reading excel files. **Support for binary file handles in `to_csv`**

`to_csv()` supports file handles in binary mode (GH19827 and GH35058) with encoding (GH13068 and GH23854) and compression (GH22555). If does not automatically detect whether the file handle is opened in binary or text mode, it is necessary to provide `mode="wb"`.

For example:

```
In [1]: import io
```

```
In [2]: data = pd.DataFrame([0, 1, 2])
```

```
In [3]: buffer = io.BytesIO()
```

In [4]: data.to_csv(buffer, encoding="utf-8", compression="gzip")
Support for short caption and table position in to_latex

DataFrame.to_latex() now allows one to specify a floating table position (GH35281) and a short caption (GH36267).

The keyword position has been added to set the position.

In [5]: data = pd.DataFrame({'a': [1, 2], 'b': [3, 4]}) **In [6]:** table = data.to_latex(position='ht')

In [7]: print(table)
`\begin{table}[ht]`
`\centering`
`\begin{tabular}{lr}`
`\toprule`
`{ } & a & b \\\`
`\midrule`
`& 1 & 3 \\\`
`& 2 & 4 \\\`
`\bottomrule`
`\end{tabular}`
`\end{table}`

Usage of the keyword caption has been extended. Besides taking a single string as an argument, one can optionally provide a tuple (full_caption, short_caption) to add a short caption macro.

In [8]: data = pd.DataFrame({'a': [1, 2], 'b': [3, 4]}) **In [9]:** table = data.to_latex(caption=('the full long caption', 'short caption'))

In [10]: print(table)
`\begin{table}`
`\centering`
`\caption[short caption]{the full long caption}`
`\begin{tabular}{lr}`
`\toprule`
`{ } & a & b \\\`
`\midrule`
`& 1 & 3 \\\`

& 2 & 4 \\
\bottomrule
\end{tabular}
\end{table}

Change in default floating precision for `read_csv` and `read_table`

For the C parsing engine, the methods `read_csv()` and `read_table()` previously defaulted to a parser that could read floating point numbers slightly incorrectly with respect to the last bit in precision. The option `floating_precision="high"` has always been available to avoid this issue. Beginning with this version, the default is now to use the more accurate parser by making `floating_precision=None` correspond to the high precision parser, and the new option `floating_precision="legacy"` to use the legacy parser. The change to using the higher precision parser by default should have no impact on performance. (GH17154)

Experimental nullable data types for float data

We've added `Float32Dtype` / `Float64Dtype` and `FloatingArray`. These are extension data types dedicated to floating point data that can hold the `pd.NA` missing value indicator (GH32265, GH34307).

While the default float data type already supports missing values using `np.nan`, these new data types use `pd.NA` (and its corresponding behavior) as the missing value indicator, in line with the already existing nullable *integer* and *boolean* data types.

One example where the behavior of `np.nan` and `pd.NA` is different is comparison operations:

```
# the default NumPy float64 dtype
In [11]: s1 = pd.Series([1.5, None])
In [12]: s1 Out[12]:
1.5
NaN
Length: 2, dtype: float64
In [13]: s1 > 1
Out[13]:
```

```
True
False
Length: 2, dtype: bool
# the new nullable float64 dtype
In [14]: s2 = pd.Series([1.5, None], dtype="Float64")
In [15]: s2 Out[15]:
1.5
<NA>
Length: 2, dtype: Float64
In [16]: s2 > 1
Out[16]:
True
<NA>
Length: 2, dtype: boolean
```

See the *Experimental NA scalar to denote missing values* doc section for more details on the behavior when using the `pd.NA` missing value indicator.

As shown above, the dtype can be specified using the “Float64” or “Float32” string (capitalized to distinguish it from the default “float64” data type). Alternatively, you can also use the dtype object:

```
In [17]: pd.Series([1.5, None], dtype=pd.Float32Dtype()) Out[17]:
1.5
<NA>
Length: 2, dtype: Float32
```

Operations with the existing integer or boolean nullable data types that give float results will now also use the nullable floating data types (GH38178).
Warning: Experimental: the new floating data types are currently experimental, and their behavior or API may still change without warning. Especially the behavior regarding NaN (distinct from NA missing values) is subject to change.

Index/column name preservation when aggregating

When aggregating using `concat()` or the `DataFrame` constructor, will now attempt to preserve index and column names whenever possible (GH35847).

In the case where all inputs share a common name, this name will be assigned to the result. When the input names do not all agree, the result will be unnamed. Here is an example where the index name is preserved:

```
In [18]: idx = pd.Index(range(5), name='abc')
```

```
In [19]: ser = pd.Series(range(5, 10), index=idx)
```

```
In [20]: pd.concat({'x': ser[1:], 'y': ser[:-1]}, axis=1) Out[20]:
```

```
x y
abc
NaN 5.0
6.0 6.0
7.0 7.0
8.0 8.0
9.0 NaN
[5 rows x 2 columns]
```

The same is true for *MultiIndex*, but the logic is applied separately on a level-by-level basis.

GroupBy supports EWM operations directly

DataFrameGroupBy now supports exponentially weighted window operations directly (GH16037).

```
In [21]: df = pd.DataFrame({'A': ['a', 'b', 'a', 'b'], 'B': range(4)})
```

```
In [22]: df Out[22]:
```

```
A B
a 0
b 1
a 2
b 3
[4 rows x 2 columns]
```

```
In [23]: df.groupby('A').ewm(com=1.0).mean() Out[23]:
```

```
B
A
a 0 0.000000
1.333333b 1 1.000000
2.333333
(continues on next page)
```

Version

(continued from previous page)

[4 rows x 1 columns]

Additionally mean supports execution via Numba with the engine and engine_kwargs arguments. Numba must be installed as an optional dependency to use this feature. **Other enhancements**

Added day_of_week (compatibility alias dayofweek) property to *Timestamp*, *DatetimeIndex*, *Period*, *PeriodIndex* (GH9605)

Added day_of_year (compatibility alias dayofyear) property to *Timestamp*, *DatetimeIndex*, *Period*, *PeriodIndex* (GH9605)

Added *set_flags()* for setting table-wide flags on a Series or DataFrame (GH28394) *DataFrame.applymap()* now supports na_action (GH23803) *Index* with object dtype supports division and multiplication (GH34160)

io.sql.get_schema() now supports a schema keyword argument that will add a schema into the create table statement (GH28486)

DataFrame.explode() and *Series.explode()* now support exploding of sets (GH35614) *DataFrame.hist()* now supports time series (datetime) data (GH32590)

Styler.set_table_styles() now allows the direct styling of rows and columns and can be chained (GH35607)

Styler now allows direct CSS class name addition to individual data cells (GH36159) *Rolling.mean()* and *Rolling.sum()* use Kahan summation to calculate the mean to avoid numerical problems (GH10319, GH11645, GH13254, GH32761, GH36031)

DatetimeIndex.searchsorted(), *TimedeltaIndex.searchsorted()*, *PeriodIndex.searchsorted()*, and *Series.searchsorted()* with datetime-like dtypes will now try to cast string arguments (list-like and scalar) to the matching datetime-like type (GH36346)

Added methods *IntegerArray.prod()*, *IntegerArray.min()*, and *IntegerArray.max()* (GH33790)

Calling a NumPy ufunc on a DataFrame with extension types now preserves

the extension types when possible (GH23743)

Calling a binary-input NumPy ufunc on multiple DataFrame objects now aligns, matching the behavior of binary operations and ufuncs on Series (GH23743). This change has been reverted in 1.2.1, and the behaviour to not align DataFrames is deprecated instead, see the *the 1.2.1 release notes*.

Where possible `RangeIndex.difference()` and `RangeIndex.symmetric_difference()` will return *RangeIndex* instead of *Int64Index* (GH36564)

`DataFrame.to_parquet()` now supports *MultiIndex* for columns in parquet format (GH34777)

`read_parquet()` gained a `use_nullable_dtypes=True` option to use nullable dtypes that use `pd.NA` as missing value indicator where possible for the resulting DataFrame (default is `False`, and only applicable for `engine="pyarrow"`) (GH31242)

Added `Rolling.sem()` and `Expanding.sem()` to compute the standard error of the mean (GH26476)

`Rolling.var()` and `Rolling.std()` use Kahan summation and Welford's Method to avoid numerical issues (GH37051)

`DataFrame.corr()` and `DataFrame.cov()` use Welford's Method to avoid numerical issues (GH37448)

`DataFrame.plot()` now recognizes `xlabel` and `ylabel` arguments for plots of type `scatter` and `hexbin` (GH37001)

DataFrame now supports the `divmod` operation (GH37165)

`DataFrame.to_parquet()` now returns a bytes object when no path argument is passed (GH37105)

`Rolling` now supports the `closed` argument for fixed windows (GH34315)

DatetimeIndex and *Series* with `datetime64` or `datetime64tz` dtypes now support `std` (GH37436)

`Window` now supports all Scipy window types in `win_type` with flexible keyword argument support (GH34556)

`testing.assert_index_equal()` now has a `check_order` parameter that allows

indexes to be checked in an order-insensitive manner (GH37478)
read_csv() supports memory-mapping for compressed files (GH37621) Add support for *min_count* keyword for *DataFrame.groupby()* and *DataFrame.resample()* for functions *min*, *max*, *first* and *last* (GH37821, GH37768) Improve error reporting for *DataFrame.merge()* when invalid merge column definitions were given (GH16228)
Improve numerical stability for *Rolling.skew()*, *Rolling.kurt()*, *Expanding.skew()* and *Expanding.kurt()* through implementation of Kahan summation (GH6929) Improved error reporting for subsetting columns of a *DataFrameGroupBy* with *axis=1* (GH37725)
Implement method *cross* for *DataFrame.merge()* and *DataFrame.join()* (GH5401) When *read_csv()*, *read_sas()* and *read_json()* are called with *chunksize/iterator* they can be used in a *with* statement as they return context-managers (GH38225) Augmented the list of named colors available for styling Excel exports, enabling all of CSS4 colors (GH38247)

Notable bug fixes

These are bug fixes that might have notable behavior changes.

Consistency of DataFrame Reductions

DataFrame.any() and *DataFrame.all()* with *bool_only=True* now determines whether to exclude object-dtype columns on a column-by-column basis, instead of checking if *all* object-dtype columns can be considered boolean.

This prevents pathological behavior where applying the reduction on a subset of columns could result in a larger Series result. See (GH37799).

In [24]: `df = pd.DataFrame({"A": ["foo", "bar"], "B": [True, False]}, dtype=object)` **In [25]:** `df["C"] = pd.Series([True, True])`

Previous behavior :

5.1. Version 1.2

In [5]: `df.all(bool_only=True)` **Out[5]:**

C True

dtype: bool

In [6]: `df[["B", "C"]].all(bool_only=True)` **Out[6]:**

B False C True

dtype: bool

New behavior:

In [26]: In [5]: df.all(bool_only=True) **Out[26]:**

False

True

Length: 2, dtype: bool

In [27]: In [6]: df[["B", "C"]].all(bool_only=True) **Out[27]:**

False

True

Length: 2, dtype: bool

Other DataFrame reductions with numeric_only=None will also avoid this pathological behavior (GH37827):

In [28]: df = pd.DataFrame({"A": [0, 1, 2], "B": ["a", "b", "c"]},
dtype=object) *Previous behavior:*

In [3]: df.mean()

Out[3]: Series([], dtype: float64)

In [4]: df[["A"]].mean() **Out[4]:**

A 1.0

dtype: float64

New behavior:

In [29]: df.mean() **Out[29]:**

A 1.0

Length: 1, dtype: float64

In [30]: df[["A"]].mean() **Out[30]:**

A 1.0

Length: 1, dtype: float64

Moreover, DataFrame reductions with numeric_only=None will now be consistent with their Series counterparts. In particular, for reductions where the Series method raises TypeError, the DataFrame reduction will now consider that column non-numeric instead of casting to a NumPy array which may have different semantics (GH36076, GH28949, GH21020).

In [31]: ser = pd.Series([0, 1], dtype="category", name="A") **In [32]:** df =
ser.to_frame() *Previous behavior:*

In [5]: df.any() **Out[5]:**

A True

dtype: bool

New behavior:

In [33]: df.any()

Out[33]: Series([], Length: 0, dtype: bool)

Increased minimum version for Python

1.2.0 supports Python 3.7.1 and higher (GH35214).

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH35214). If installed, we now require: Package

numpy pytz

python-dateutil Minimum Version

1.16.5

2017.3

2.7.3

bottleneck 1.2.1 Required Changed

X X X X X

numexpr 2.6.8

X pytest 5.0.1 X (dev)

mypy 0.782 X
(dev)

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of . Optional libraries below the lowest tested version may still work, but are not considered supported.

5.1. Version 1.2 Package Minimum Changed Version

beautifulsoup4 4.6.0 fastparquet 0.3.2 fsspec 0.7.4 gcsfs 0.6.0 lxml 4.3.0 X

matplotlib 2.2.3 X numba 0.46.0 openpyxl 2.6.0 X pyarrow 0.15.0 X

pymysql 0.7.11 X pytables 3.5.1 X s3fs 0.4.0 scipy 1.2.0 sqlalchemy 1.2.8 X

xarray 0.12.3 X xlrd 1.2.0 X xlswriter 1.0.2 X xlwt 1.3.0 X

-gbq 0.12.0

See *Dependencies* and *Optional dependencies* for more.

Other API changes

Sorting in descending order is now stable for *Series.sort_values()* and *Index.sort_values()* for Datetime-like *Index* subclasses. This will affect sort order when sorting a DataFrame on multiple columns, sorting with a key function that produces duplicates, or requesting the sorting index when using *Index.sort_values()*. When using *Series.value_counts()*, the count of missing values is no longer necessarily last in the list of duplicate counts. Instead, its position corresponds to the position in the original Series. When using *Index.sort_values()* for Datetime-like *Index* subclasses, NaTs ignored the *na_position* argument and were sorted to the beginning. Now they respect *na_position*, the default being last, same as other *Index* subclasses (GH35992)

Passing an invalid *fill_value* to *Categorical.take()*, *DatetimeArray.take()*, *TimedeltaArray.take()*, or *PeriodArray.take()* now raises a *TypeError* instead of a *ValueError* (GH37733)

Passing an invalid *fill_value* to *Series.shift()* with a *CategoricalDtype* now raises a *TypeError* instead of a *ValueError* (GH37733)

Passing an invalid value to *IntervalIndex.insert()* or *CategoricalIndex.insert()* now raises a *TypeError* instead of a *ValueError* (GH37733)

Attempting to reindex a Series with a *CategoricalIndex* with an invalid *fill_value* now raises a *TypeError* instead of a *ValueError* (GH37733)

CategoricalIndex.append() with an index that contains non-category values will now cast instead of raising *TypeError* (GH38098)

Deprecations

Deprecated parameter *inplace* in *MultiIndex.set_codes()* and *MultiIndex.set_levels()* (GH35626)

Deprecated parameter *dtype* of method *copy()* for all *Index* subclasses. Use the *astype()* method instead for changing dtype (GH35853)

Deprecated parameters *levels* and *codes* in *MultiIndex.copy()*. Use the *set_levels()* and *set_codes()* methods instead (GH36685)

Date parser functions `parse_date_time()`, `parse_date_fields()`, `parse_all_fields()` and `generic_parser()` from `.io.date_converters` are deprecated and will be removed in a future version; use `to_datetime()` instead (GH35741)

`DataFrame.lookup()` is deprecated and will be removed in a future version, use `DataFrame.melt()` and `DataFrame.loc()` instead (GH35224)

The method `Index.to_native_types()` is deprecated. Use `.astype(str)` instead (GH28867) Deprecated indexing `DataFrame` rows with a single datetime-like string as `df[string]` (given the ambiguity whether it is indexing the rows or selecting a column), use `df.loc[string]` instead (GH36179)

Deprecated `Index.is_all_dates()` (GH27744)

The default value of `regex` for `Series.str.replace()` will change from `True` to `False` in a future release. In addition, single character regular expressions will *not* be treated as literal strings when `regex=True` is set (GH24804)

Deprecated automatic alignment on comparison operations between `DataFrame` and `Series`, do `frame, ser = frame.align(ser, axis=1, copy=False)` before e.g. `frame == ser` (GH28759)

`Rolling.count()` with `min_periods=None` will default to the size of the window in a future version (GH31302)

Using “outer” ufuncs on `DataFrames` to return 4d ndarray is now deprecated. Convert to an ndarray first (GH23743)

Deprecated slice-indexing on tz-aware `DatetimeIndex` with naive datetime objects, to match scalar indexing behavior (GH36148)

`Index.ravel()` returning a `np.ndarray` is deprecated, in the future this will return a view on the same index (GH19956)

Deprecate use of strings denoting units with ‘M’, ‘Y’ or ‘y’ in `to_timedelta()` (GH36666)

`Index` methods `&`, `|`, and `^` behaving as the set operations `Index.intersection()`, `Index.union()`, and `Index.symmetric_difference()`, respectively, are deprecated and in the future will behave as pointwise boolean operations matching `Series` behavior. Use the named set methods instead (GH36758)

`Categorical.is_dtype_equal()` and `CategoricalIndex.is_dtype_equal()` are deprecated, will be removed in a future version (GH37545)

`Series.slice_shift()` and `DataFrame.slice_shift()` are deprecated, use `Series.shift()` or `DataFrame.shift()` instead (GH37601)

Partial slicing on unordered `DatetimeIndex` objects with keys that are not in the index is deprecated and will be removed in a future version (GH18531)

The `how` keyword in `PeriodIndex.astype()` is deprecated and will be removed in a future version, use `index.to_timestamp(how=how)` instead (GH37982)

Deprecated `Index.asi8()` for `Index` subclasses other than `DatetimeIndex`, `TimedeltaIndex`, and `PeriodIndex` (GH37877)

. Version 1.2

The inplace parameter of `Categorical.remove_unused_categories()` is deprecated and will be removed in a future version (GH37643)

The `null_counts` parameter of `DataFrame.info()` is deprecated and replaced by `show_counts`. It will be removed in a future version (GH37999)

Calling NumPy ufuncs on non-aligned DataFrames

Calling NumPy ufuncs on non-aligned DataFrames changed behaviour in 1.2.0 (to align the inputs before calling the ufunc), but this change is reverted in 1.2.1. The behaviour to not align is now deprecated instead, see the *the 1.2.1 release notes* for more details.

Performance improvements

Performance improvements when creating `DataFrame` or `Series` with dtype `str` or `StringDtype` from array with many string elements (GH36304, GH36317, GH36325, GH36432, GH37371)

Performance improvement in `GroupBy.agg()` with the numba engine (GH35759) Performance improvements when creating `Series.map()` from a huge dictionary (GH34717)

Performance improvement in `GroupBy.transform()` with the numba engine (GH36240) `Styler` `uuid` method altered to compress data transmission over web whilst maintaining reasonably low table collision probability (GH36345)

Performance improvement in `to_datetime()` with non-ns time unit for floatdtype columns (GH20445) • Performance improvement in setting values on an `IntervalArray` (GH36310)

The internal index method `_shallow_copy()` now makes the new index and original index share cached attributes, avoiding creating these again, if created on either. This can speed up operations that depend on creating copies of existing indexes (GH36840)

Performance improvement in `RollingGroupby.count()` (GH35625)
Small performance decrease to `Rolling.min()` and `Rolling.max()` for fixed windows (GH36567)

Reduced peak memory usage in `DataFrame.to_pickle()` when using `protocol=5` in python 3.8+ (GH34244)

Faster `dir` calls when the object has many index labels, e.g. `dir(ser)` (GH37450)
Performance improvement in `ExpandingGroupby` (GH37064)
Performance improvement in `Series.astype()` and `DataFrame.astype()` for *Categorical*

Performance improvement in `DataFrame.groupby()` for float dtype (GH28303), changes of the underlying hash-function can lead to changes in float based indexes sort ordering for ties (e.g. `Index.value_counts()`)

Performance improvement in `pd.isin()` for inputs with more than 1e6 elements (GH36611)

Performance improvement for `DataFrame.__setitem__()` with list-like indexers (GH37954)
`read_json()` now avoids reading entire file into memory when chunksize is specified (GH34548)

Bug fixes

Categorical

`Categorical.fillna()` will always return a copy, validate a passed fill value regardless of whether there are any NAs to fill, and disallow an `NaT` as a fill value for numeric categories (GH36530)

Bug in `Categorical.__setitem__()` that incorrectly raised when trying to set a tuple value (GH20439) • Bug in `CategoricalIndex.equals()` incorrectly casting non-category entries to `np.nan` (GH37667)

Bug in `CategoricalIndex.where()` incorrectly setting non-category entries to `np.nan` instead of raising `TypeError` (GH37977)

Bug in `Categorical.to_numpy()` and `np.array(categorical)` with tz-aware `datetime64` categories incorrectly dropping the time zone information instead of casting to object dtype (GH38136)

Datetime-like

Bug in `DataFrame.combine_first()` that would convert datetime-like column on other `DataFrame` to integer when the column is not present in original `DataFrame` (GH28481) Bug in `DatetimeArray.date` where a `ValueError` would be raised with a read-only backing array (GH33530)

Bug in NaT comparisons failing to raise `TypeError` on invalid inequality comparisons (GH35046)

Bug in `DateOffset` where attributes reconstructed from pickle files differ from original objects when input values exceed normal ranges (e.g. `months=12`) (GH34511) Bug in `DatetimeIndex.get_slice_bound()` where `datetime.date` objects were not accepted or naive `Timestamp` with a tz-aware `DatetimeIndex` (GH35690)

Bug in `DatetimeIndex.slice_locs()` where `datetime.date` objects were not accepted Bug in `DatetimeIndex.searchsorted()`, `TimedeltaIndex.searchsorted()`, `PeriodIndex.searchsorted()`, and `Series.searchsorted()` with `datetime64`, `timedelta64` or `Period` dtype placement of NaT values being inconsistent with NumPy (GH36176, GH36254)

Inconsistency in `DatetimeArray`, `TimedeltaArray`, and `PeriodArray` method `__setitem__` casting arrays of strings to datetime-like scalars but not scalar strings (GH36261) Bug in `DatetimeArray.take()` incorrectly allowing `fill_value` with a mismatched time zone (GH37356)

Bug in `DatetimeIndex.shift` incorrectly raising when shifting empty indexes (GH14811)

`Timestamp` and `DatetimeIndex` comparisons between tz-aware and tz-naive objects now follow the standard library datetime behavior, returning `True/False` for `!=`/`==` and raising for inequality comparisons (GH28507)

Bug in `DatetimeIndex.equals()` and `TimedeltaIndex.equals()` incorrectly considering int64 indexes as equal (GH36744)
`Series.to_json()`, `DataFrame.to_json()`, and `read_json()` now implement time zone parsing when orient structure is table (GH35973)
`astype()` now attempts to convert to `datetime64[ns, tz]` directly from object with inferred time zone from
Bug in `TimedeltaIndex.sum()` and `Series.sum()` with `timedelta64` dtype on an empty index or series returning `NaT` instead of `Timedelta(0)` (GH31751)
Bug in `DatetimeArray.shift()` incorrectly allowing `fill_value` with a mismatched time zone (GH37299)

Bug in adding a *BusinessDay* with nonzero offset to a non-scalar other (GH37457) Bug in `to_datetime()` with a read-only array incorrectly raising (GH34857)

Bug in `Series.isin()` with `datetime64[ns]` dtype and `DatetimeIndex.isin()` incorrectly casting integers to datetimes (GH36621)
Bug in `Series.isin()` with `datetime64[ns]` dtype and `DatetimeIndex.isin()` failing to consider tz-aware and tz-naive datetimes as always different (GH35728) Bug in `Series.isin()` with `PeriodDtype` dtype and `PeriodIndex.isin()` failing to consider arguments with different `PeriodDtype` as always different (GH37528)
Bug in *Period* constructor now correctly handles nanoseconds in the value argument (GH34621 and GH17053)

Timedelta

Bug in *TimedeltaIndex*, *Series*, and *DataFrame* floor-division with `timedelta64` dtypes and `NaT` in the denominator (GH35529)
Bug in parsing of ISO 8601 durations in *Timedelta* and `to_datetime()` (GH29773, GH36204)
Bug in `to_timedelta()` with a read-only array incorrectly raising (GH34857)
Bug in *Timedelta* incorrectly truncating to sub-second portion of a string input when it has precision higher than nanoseconds (GH36738)

Timezones

Bug in `date_range()` was raising `AmbiguousTimeError` for valid input with `ambiguous=False` (GH35297)

Bug in *Timestamp.replace()* was losing fold information (GH37610)

Numeric

Bug in *to_numeric()* where float precision was incorrect (GH31364)

Bug in *DataFrame.any()* with *axis=1* and *bool_only=True* ignoring the *bool_only* keyword (GH32432)

Bug in *Series.equals()* where a *ValueError* was raised when NumPy arrays were compared to scalars (GH35267)

Bug in *Series* where two *Series* each have a *DatetimeIndex* with different time zones having those indexes incorrectly changed when performing arithmetic operations (GH33671)

Bug in *.testing* module functions when used with *check_exact=False* on complex numeric types (GH28235)

Bug in *DataFrame.__rmatmul__()* error handling reporting transposed shapes
Bug in *Series* flex arithmetic methods where the result when operating with a list, tuple or np.

ndarray would have an incorrect name (GH36760)

Bug in *IntegerArray* multiplication with *timedelta* and *np.timedelta64* objects (GH36870)

Bug in *MultiIndex* comparison with tuple incorrectly treating tuple as array-like (GH21517)

Bug in *DataFrame.diff()* with *datetime64* dtypes including *NaT* values failing to fill *NaT* results correctly (GH32441)

Bug in *DataFrame* arithmetic ops incorrectly accepting keyword arguments (GH36843) Bug in *IntervalArray* comparisons with *Series* not returning *Series* (GH36908)

Bug in *DataFrame* allowing arithmetic operations with list of array-likes with undefined results. Behavior changed to raising *ValueError* (GH36702)

Bug in *DataFrame.std()* with *timedelta64* dtype and *skipna=False* (GH37392) Bug in *DataFrame.min()* and *DataFrame.max()* with *datetime64* dtype and *skipna=False* (GH36907)

Bug in *DataFrame.idxmax()* and *DataFrame.idxmin()* with mixed dtypes incorrectly raising

TypeError (GH38195)

Conversion

Bug in *DataFrame.to_dict()* with orient='records' now returns python native datetime objects for datetime-like columns (GH21256)

Bug in *Series.astype()* conversion from string to float raised in presence of pd.NA values

(GH37626)

Strings

Bug in *Series.to_string()*, *DataFrame.to_string()*, and *DataFrame.to_latex()* adding a leading space when index=False (GH24980)

Bug in *to_numeric()* raising a TypeError when attempting to convert a string dtype Series containing only numeric strings and NA (GH37262)

Interval

Bug in *DataFrame.replace()* and *Series.replace()* where *Interval* dtypes would be converted to object dtypes (GH34871)

Bug in *IntervalIndex.take()* with negative indices and fill_value=None (GH37330) Bug in *IntervalIndex.putmask()* with to object dtype (GH37968) datetime-like dtype incorrectly casting Bug in *IntervalArray.astype()* incorrectly dropping dtype information with a *CategoricalDtype* object

Indexing

Bug in *PeriodIndex.get_loc()* incorrectly raising ValueError on non-datelike strings instead of KeyError, causing similar errors in *Series.__getitem__()*, *Series.__contains__()*, and *Series.loc.__getitem__()* (GH34240)

Bug in *Index.sort_values()* where, when empty values were passed, the method would break by trying to compare missing values instead of pushing them to the end of the sort order (GH35584)

Bug in *Index.get_indexer()* and *Index.get_indexer_non_unique()* where int64 arrays are returned instead of intp (GH36359)

Bug in *DataFrame.sort_index()* where parameter ascending passed as a list

on a single level index gives wrong result (GH32334)

Bug in *DataFrame.reset_index()* was incorrectly raising a `ValueError` for input with a *MultiIndex* with missing values in a level with Categorical dtype (GH24206) Bug in indexing with boolean masks on datetime-like values sometimes returning a view instead of a copy (GH36210)

Bug in *DataFrame.__getitem__()* and *DataFrame.loc.__getitem__()* with *IntervalIndex* columns and a numeric indexer (GH26490)

Bug in *Series.loc.__getitem__()* with a non-unique *MultiIndex* and an empty-list indexer (GH13691)

Bug in indexing on a *Series* or *DataFrame* with a *MultiIndex* and a level named "0" (GH37194)

Bug in *Series.__getitem__()* when using an unsigned integer array as an indexer giving incorrect results or segfaulting instead of raising `KeyError` (GH37218)

Bug in *Index.where()* incorrectly casting numeric values to strings (GH37591) Bug in *DataFrame.loc()* returning empty result when indexer is a slice with negative step size (GH38071)

Bug in *Series.loc()* and *DataFrame.loc()* raises when the index was of object dtype and the given numeric label was in the index (GH26491)

Bug in *DataFrame.loc()* returned requested key plus missing values when loc was applied to single level from a *MultiIndex* (GH27104)

Bug in indexing on a *Series* or *DataFrame* with a *CategoricalIndex* using a list-like indexer containing NA values (GH37722)

Bug in *DataFrame.loc.__setitem__()* expanding an empty *DataFrame* with mixed dtypes (GH37932)

Bug in *DataFrame.xs()* ignored `droplevel=False` for columns (GH19056) Bug in *DataFrame.reindex()* raising `IndexingError` wrongly for empty *DataFrame* with `tolerance` not `None` or `method="nearest"` (GH27315)

Bug in indexing on a *Series* or *DataFrame* with a *CategoricalIndex* using list-like indexer that contains elements that are in the index's categories but not in the index itself failing to raise `KeyError` (GH37901)

Bug on inserting a boolean label into a *DataFrame* with a numeric *Index*

columns incorrectly casting to integer (GH36319)

Bug in *DataFrame.iloc()* and *Series.iloc()* aligning objects in `__setitem__` (GH22046) Bug in *MultiIndex.drop()* does not raise if labels are partially found (GH37820)

Bug in *DataFrame.loc()* did not raise *KeyError* when missing combination was given with `slice(None)` for remaining levels (GH19556)

Bug in *DataFrame.loc()* raising *TypeError* when non-integer slice was given to select values from *MultiIndex* (GH25165, GH24263)

Bug in *Series.at()* returning *Series* with one element instead of scalar when index is a *MultiIndex* with one level (GH38053)

Bug in *DataFrame.loc()* returning and assigning elements in wrong order when indexer is differently ordered than the *MultiIndex* to filter (GH31330, GH34603)

Bug in *DataFrame.loc()* and *DataFrame.__getitem__()* raising *KeyError* when columns were *MultiIndex* with only one level (GH29749)

Bug in *Series.__getitem__()* and *DataFrame.__getitem__()* raising blank *KeyError* without missing keys for *IntervalIndex* (GH27365)

Bug in setting a new label on a *DataFrame* or *Series* with a *CategoricalIndex* incorrectly raising *TypeError* when the new label is not among the index's categories (GH38098) Bug in *Series.loc()* and *Series.iloc()* raising

ValueError when inserting a list-like `np.array`, list or tuple in an object *Series* of equal length (GH37748, GH37486) Bug in *Series.loc()* and *Series.iloc()*

setting all the values of an object *Series* with those of a list-like *ExtensionArray* instead of inserting it (GH38271)

Missing

Bug in *SeriesGroupBy.transform()* now correctly handles missing values for `dropna=False` (GH35014)

Bug in *Series.nunique()* with `dropna=True` was returning incorrect results when both NA and None missing values were present (GH37566)

Bug in *Series.interpolate()* where kwarg `limit_area` and `limit_direction` had no effect when using methods `pad` and `backfill` (GH31048)

MultiIndex

Bug in *DataFrame.xs()* when used with *IndexSlice* raises *TypeError* with message "Expected label or tuple of labels" (GH35301)

Bug in *DataFrame.reset_index()* with NaT values in index raises *ValueError*

with message

"cannot convert float NaN to integer" (GH36541)

Bug in *DataFrame.combine_first()* when used with *MultiIndex* containing string and NaN values raises *TypeError* (GH36562)

Bug in *MultiIndex.drop()* dropped NaN values when non existing key was given as input (GH18853)

Bug in *MultiIndex.drop()* dropping more values than expected when index has duplicates and is not sorted (GH33494)

5.1. Version 1.2

I/O

read_sas() no longer leaks resources on failure (GH35566)

Bug in *DataFrame.to_csv()* and *Series.to_csv()* caused a *ValueError* when it was called with a filename in combination with mode containing a b (GH35058)

Bug in *read_csv()* with *float_precision='round_trip'* did not handle decimal and thousands parameters (GH35365)

to_pickle() and *read_pickle()* were closing user-provided file objects (GH35679) *to_csv()* passes compression arguments for 'gzip' always to *gzip.GzipFile* (GH28103)

to_csv() did not support zip compression for binary file object not having a filename (GH35058)

to_csv() and *read_csv()* did not honor compression and encoding for path-like objects that are internally converted to file-like objects (GH35677, GH26124, GH32392)

DataFrame.to_pickle(), *Series.to_pickle()*, and *read_pickle()* did not support compression for file-objects (GH26237, GH29054, GH29570)

Bug in *LongTableBuilder.middle_separator()* was duplicating LaTeX longtable entries in the List of Tables of a LaTeX document (GH34360)

Bug in *read_csv()* with *engine='python'* truncating data if multiple items present in first row and first element started with BOM (GH36343)

Removed *private_key* and *verbose* from *read_gbq()* as they are no longer supported in *gbq* (GH34654, GH30200)

Bumped minimum *pytables* version to 3.5.1 to avoid a *ValueError* in *read_hdf()* (GH24839)

Bug in *read_table()* and *read_csv()* when *delim_whitespace=True* and *sep=default* (GH36583)

Bug in *DataFrame.to_json()* and *Series.to_json()* when used with *lines=True* and *orient='records'* the last line of the record is not appended with 'new line character' (GH36888)

Bug in *read_parquet()* with fixed offset time zones. String representation of time zones was not recognized (GH35997, GH36004)

Bug in *DataFrame.to_html()*, *DataFrame.to_string()*, and *DataFrame.to_latex()* ignoring the *na_rep* argument when *float_format* was also specified (GH9046, GH13828) Bug in output rendering of complex numbers showing too many trailing zeros (GH36799)

Bug in HDFStore threw a *TypeError* when exporting an empty *DataFrame* with *datetime64[ns, tz]* dtypes with a fixed HDF5 store (GH20594)

Bug in HDFStore was dropping time zone information when exporting a *Series* with *datetime64[ns, tz]* dtypes with a fixed HDF5 store (GH20594) *read_csv()* was closing user-provided binary file handles when *engine="c"* and an encoding was requested (GH36980)

Bug in *DataFrame.to_hdf()* was not dropping missing rows with *dropna=True* (GH35719)

Bug in *read_html()* was raising a *TypeError* when supplying a *pathlib.Path* argument to the *io* parameter (GH37705)

DataFrame.to_excel(), *Series.to_excel()*, *DataFrame.to_markdown()*, and *Series.to_markdown()* now support writing to fsspec URLs such as S3 and Google Cloud Storage (GH33987)

Bug in *read_fwf()* with *skip_blank_lines=True* was not skipping blank lines (GH37758) Parse missing values using *read_json()* with *dtype=False* to *NaN* instead of *None* (GH28501)

read_fwf() was inferring compression with *compression=None* which was not consistent with the other *read_** functions (GH37909)

DataFrame.to_html() was ignoring *formatters* argument for *ExtensionDtype* columns (GH36525)

Bumped minimum xarray version to 0.12.3 to avoid reference to the removed *Panel* class (GH27101, GH37983)

DataFrame.to_csv() was re-opening file-like handles that also implement *os.PathLike* (GH38125)

Bug in the conversion of a sliced `pyarrow.Table` with missing values to a `DataFrame` (GH38525)

Bug in `read_sql_table()` raising a `sqlalchemy.exc.OperationalError` when column names contained a percentage sign (GH37517)

Period

Bug in `DataFrame.replace()` and `Series.replace()` where `Period` dtypes would be converted to object dtypes (GH34871)

Plotting

Bug in `DataFrame.plot()` was rotating xticklabels when `subplots=True`, even if the x-axis wasn't an irregular time series (GH29460)

Bug in `DataFrame.plot()` where a marker letter in the style keyword sometimes caused a `ValueError` (GH21003)

Bug in `DataFrame.plot.bar()` and `Series.plot.bar()` where ticks positions were assigned by value order instead of using the actual value for numeric or a smart ordering for string (GH26186, GH11465). This fix has been reverted in 1.2.1, see *What's new in 1.2.1 (January 20, 2021)*

Twinned axes were losing their tick labels which should only happen to all but the last row or column of 'externally' shared axes (GH33819)

Bug in `Series.plot()` and `DataFrame.plot()` was throwing a `ValueError` when the Series or DataFrame was indexed by a `TimedeltaIndex` with a fixed frequency and the x-axis lower limit was greater than the upper limit (GH37454)

Bug in `DataFrameGroupBy.boxplot()` when `subplots=False` would raise a `KeyError` (GH16748)

Bug in `DataFrame.plot()` and `Series.plot()` was overwriting matplotlib's shared y axes behavior when no `sharey` parameter was passed (GH37942)

Bug in `DataFrame.plot()` was raising a `TypeError` with `ExtensionDtype` columns (GH32073)

pandas: powerful Python data analysis toolkit, Release 1.2.3

Styler

Bug in `Styler.render()` HTML was generated incorrectly because of formatting error in `rowspan` attribute, it now matches with w3 syntax (GH38234)

Groupby/resample/rolling

Bug in *DataFrameGroupBy.count()* and *SeriesGroupBy.sum()* returning NaN for missing categories when grouped on multiple Categoricals. Now returning 0 (GH35028) Bug in *DataFrameGroupBy.apply()* that would sometimes throw an erroneous *ValueError* if the grouping axis had duplicate entries (GH16646)

Bug in *DataFrame.resample()* that would throw a *ValueError* when resampling from "D" to "24H" over a transition into daylight savings time (DST) (GH35219)

Bug when combining methods *DataFrame.groupby()* with *DataFrame.resample()* and *DataFrame.interpolate()* raising a *TypeError* (GH35325)

Bug in *DataFrameGroupBy.apply()* where a non-nuisance grouping column would be dropped from the output columns if another groupby method was called before *.apply* (GH34656)

Bug when subsetting columns on a *DataFrameGroupBy* (e.g. *df.groupby('a')[['b']]*) would reset the attributes *axis*, *dropna*, *group_keys*, *level*, *mutated*, *sort*, and *squeeze* to their default values (GH9959)

Bug in *DataFrameGroupBy.tshift()* failing to raise *ValueError* when a frequency cannot be inferred for the index of a group (GH35937)

Bug in *DataFrame.groupby()* does not always maintain column index name for any, all, bfill, ffill, shift (GH29764)

Bug in *DataFrameGroupBy.apply()* raising error with *np.nan* group(s) when *dropna=False* (GH35889)

Bug in *Rolling.sum()* returned wrong values when dtypes were mixed between float and integer and *axis=1* (GH20649, GH35596)

Bug in *Rolling.count()* returned *np.nan* with *FixedForwardWindowIndexer* as window, *min_periods=0* and only missing values in the window (GH35579)

Bug where *.core.window.Rolling* produces incorrect window sizes when using a *PeriodIndex* (GH34225)

-
-

499 Chapter 5. Release notes

Bug in *DataFrameGroupBy.ffill()* and *DataFrameGroupBy.bfill()* where a NaN group would return filled values instead of NaN when *dropna=True* (GH34725) Bug in *RollingGroupby.count()* where a *ValueError* was raised when specifying the *closed* parameter (GH35869) Bug in *DataFrameGroupBy.rolling()* returning wrong values with partial centered window (GH36040)

Bug in *DataFrameGroupBy.rolling()* returned wrong values with time aware window containing NaN. Raises *ValueError* because windows are not monotonic now (GH34617)

Bug in *Rolling.__iter__()* where a *ValueError* was not raised when *min_periods* was larger than window (GH37156)

Using *Rolling.var()* instead of *Rolling.std()* avoids numerical issues for *Rolling.corr()* when

Rolling.var() is still within floating point precision while *Rolling.std()* is not (GH31286) Bug in *DataFrameGroupBy.quantile()* and *Resampler.quantile()* raised *TypeError* when values were of type *Timedelta* (GH29485)

Bug in *Rolling.median()* and *Rolling.quantile()* returned wrong values for *BaseIndexer* subclasses with non-monotonic starting or ending points for windows (GH37153) Bug in *DataFrame.groupby()* dropped nan groups from result with *dropna=False* when grouping over a single column (GH35646, GH35542)

Bug in *DataFrameGroupBy.head()*, *DataFrameGroupBy.tail()*, *SeriesGroupBy.head()*, and *SeriesGroupBy.tail()* would raise when used with *axis=1* (GH9772) Bug in *DataFrameGroupBy.transform()* would raise when used with *axis=1* and a transformation kernel (e.g. “shift”) (GH36308)

Bug in *DataFrameGroupBy.resample()* using *.agg* with *sum* produced different result than just calling *.sum* (GH33548)

Bug in *DataFrameGroupBy.apply()* dropped values on nan group when returning the same axes with the original frame (GH38227)

Bug in *DataFrameGroupBy.quantile()* couldn't handle with arraylike *q* when grouping by columns (GH33795)

Bug in *DataFrameGroupBy.rank()* with *datetime64tz* or *period* dtype incorrectly casting results to those dtypes instead of returning *float64* dtype (GH38187)

Reshaping

Bug in `DataFrame.crosstab()` was returning incorrect results on inputs with duplicate row names, duplicate column names or duplicate names between row and column labels (GH22529)

Bug in `DataFrame.pivot_table()` with `aggfunc='count'` or `aggfunc='sum'` returning NaN for missing categories when pivoted on a Categorical. Now returning 0 (GH31422) Bug in `concat()` and `DataFrame` constructor where input index names are not preserved in some cases (GH13475)

Bug in func `crosstab()` when using multiple columns with `margins=True` and `normalize=True` (GH35144)

Bug in `DataFrame.stack()` where an empty `DataFrame.stack` would raise an error (GH36113). Now returning an empty Series with empty MultiIndex.

Bug in `Series.unstack()`. Now a Series with single level of Index trying to unstack would raise a `ValueError` (GH36113)

Bug in `DataFrame.agg()` with `func={'name':<FUNC>}` incorrectly raising `TypeError` when `DataFrame.columns==['Name']` (GH36212)

Bug in `Series.transform()` would give incorrect results or raise when the argument `func` was a dictionary (GH35811)

Bug in `DataFrame.pivot()` did not preserve *MultiIndex* level names for columns when rows and columns are both multiindexed (GH36360)

Bug in `DataFrame.pivot()` modified index argument when columns was passed but values was not (GH37635)

Bug in `DataFrame.join()` returned a non deterministic level-order for the resulting *MultiIndex*

(GH36910)

5.1. Version 1.2

Bug in `DataFrame.combine_first()` caused wrong alignment with dtype string and one level of MultiIndex containing only NA (GH37591)

Fixed regression in `merge()` on merging *DatetimeIndex* with empty `DataFrame` (GH36895)

Bug in `DataFrame.apply()` not setting index of return value when `func` return type is dict (GH37544)

Bug in `DataFrame.merge()` and `.merge()` returning inconsistent ordering in result for `how=right` and `how=left` (GH35382)

Bug in `merge_ordered()` couldn't handle list-like `left_by` or `right_by`

(GH35269) Bug in *merge_ordered()* returned wrong join result when length of *left_by* or *right_by* equals to the rows of *left* or *right* (GH38166)

Bug in *merge_ordered()* didn't raise when elements in *left_by* or *right_by* not exist in left columns or right columns (GH38167)

Bug in *DataFrame.drop_duplicates()* not validating bool dtype for *ignore_index* keyword (GH38274)

ExtensionArray

Fixed bug where *DataFrame* column set to scalar extension type via a dict instantiation was considered an object type rather than the extension type (GH35965) Fixed bug where *astype()* with equal dtype and *copy=False* would return a new object (GH28488)

Fixed bug when applying a NumPy ufunc with multiple outputs to an *IntegerArray* returning None (GH36913)

Fixed an inconsistency in *PeriodArray*'s *__init__* signature to those of *DatetimeArray* and *TimedeltaArray* (GH37289)

Reductions for *BooleanArray*, *Categorical*, *DatetimeArray*, *FloatingArray*, *IntegerArray*, *PeriodArray*, *TimedeltaArray*, and *Array* are now keyword-only methods (GH37541) Fixed a bug where a *TypeError* was wrongly raised if a membership check was made on an

ExtensionArray containing nan-like values (GH37867)

Other

Bug in *DataFrame.replace()* and *Series.replace()* incorrectly raising an *AssertionError* instead of a *ValueError* when invalid parameter combinations are passed (GH36045) Bug in *DataFrame.replace()* and *Series.replace()* with numeric values and string to *_replace* (GH34789)

Fixed metadata propagation in *Series.abs()* and ufuncs called on *Series* and *DataFrames* (GH28283)

Bug in *DataFrame.replace()* and *Series.replace()* incorrectly casting from *PeriodDtype* to object dtype (GH34871)

Fixed bug in metadata propagation incorrectly copying *DataFrame* columns as metadata when the column name overlaps with the metadata name (GH37037)

Fixed metadata propagation in the *Series.dt*, *Series.str* accessors, *DataFrame.duplicated*, *DataFrame.stack*, *DataFrame.unstack*, *DataFrame.pivot*, *DataFrame.append*, *DataFrame.diff*, *DataFrame.applymap* and *DataFrame.update* methods (GH28283, GH37381)

pandas: powerful Python data analysis toolkit, Release 1.2.3

-

-

Fixed metadata propagation when selecting columns with *DataFrame.__getitem__* (GH28283)

Bug in *Index.intersection()* with non*Index* failing to set the correct name on the returned *Index* (GH38111)

Bug in *RangeIndex.intersection()* failing to set the correct name on the returned *Index* in some corner cases (GH38197)

Bug in *Index.difference()* failing to set the correct name on the returned *Index* in some corner cases (GH38268)

Bug in *Index.union()* behaving differently depending on whether operand is an *Index* or other list-like (GH36384)

Bug in *Index.intersection()* with non-matching numeric dtypes casting to object dtype instead of minimal common dtype (GH38122)

Bug in *IntervalIndex.union()* returning an incorrectly-typed *Index* when empty (GH38282)

Passing an array with 2 or more dimensions to the *Series* constructor now raises the more specific *ValueError* rather than a bare *Exception* (GH35744)

Bug in *dir* where *dir(obj)* wouldn't show attributes defined on the instance for objects (GH37173) • Bug in *Index.drop()* raising *InvalidIndexError* when index has duplicates (GH38051)

Bug in *RangeIndex.difference()* returning *Int64Index* in some cases where it should return *RangeIndex* (GH38028)

Fixed bug in *assert_series_equal()* when comparing a datetime-like array with an equivalent non extension dtype array (GH37609)

Bug in *is_bool_dtype()* would raise when passed a valid string such as "boolean" (GH38386)

Fixed regression in of *DataFrame* are logical operators raising `ValueError` when columns a

CategoricalIndex with unused categories (GH38367)

-
-
-

5.1. Version 1.2 504

Contributors

A total of 257 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

21CSM +
AbdulMAbdi +
Abhiraj Hinge +
Abhishek Mangla +
Abo7atm +
Adam Spannbauer +
Albert Villanova del Moral
Alex Kirko
Alex Lim +
Alex Thorne +
Aleš Erjavec +
Ali McMaster
Amanda Dsouza +
Amim Knabben +
Andrew Wieteska
Anshoo Rajput +
Anthony Milbourne
Arun12121 +
Asish Mahapatra
Avinash Pancham +
BeanNan +

-
-
-

Ben Forbes +
Brendan Wilby +
Bruno Almeida +
Byron Boulton +
Chankey Pathak
Chris Barnes +
Chris Lynch +
Chris Withers
Christoph Deil +
Christopher Hadley +
Chuanzhu Xu
Coelhudo +
Dan Moore
Daniel Saxton
David Kwong +
David Li +
David Mrva +
Deepak Pandey +
Deepyaman Datta
Devin Petersohn
Dmitriy Perepelkin +
Douglas Hanley +
Dags Gr⁻inbergs + Eli Treuherz + Elliot Rampono + Erfan Nariman

-
-
-

Eric Goddard
Eric Leung +
Eric Wieser
Ethan Chen +
Eve +
Eyal Trabelsi +
Fabian Gebhart +
Fangchen Li
Felix Claessen +

Finlay Maguire +
Florian Roscheck + Gabriel Monteiro
Gautham +
Gerard Jorgensen + Gregory Livschitz
Hans
Harsh Sharma
Honfung Wong +
Igor Gotlibovych + Iqrar Agalosi Nureyza Irv Lustig
Isaac Virshup
Jacob Peacock
Jacob Stevens-Haas +

-
-
-

Jan Müller +
Janus
Jeet Parekh
Jeff Hernandez +
Jeff Reback
Jiaxiang
Joao Pedro Berno Zanutto +
Joel Nothman
Joel Whittier + John Karasinski + John McGuigan + Johnny Pribyl +
Jonas Laursen +
Jonathan Shreckengost +
Joris Van den Bossche
Jose +
JoseNavy +
Josh Temple +
Jun Kudo +
Justin Essert
Justin Sexton +
Kaiqi Dong
Kamil Trocewicz +
Karthik Mathur

Kashif +
Kenny Huynh

-
-
-

Kevin Sheppard
Kumar Shivam +
Leonardus Chen +
Levi Matus +
Lucas Rodés-Guirao +
Luis Pinto + • Lynch +
Marc Garcia
Marco Gorelli
Maria-Alexandra Ilie +
Marian Denes
Mark Graham +
Martin Durant
Matt Roeschke
Matthew Roeschke
Matthias Bussonnier
Maxim Ivanov +
Mayank Chaudhary +
MeeseeksMachine Meghana Varanasi + Metehan Kutlu + Micael Jarniac +
Micah Smith +
Michael Marino
Miroslav Šedivý
Mohammad Jafar Mashhadi
Mohammed Kashif +

-
-
-

Nagesh Kumar C +
Nidhi Zare +
Nikhil Choudhary +

Number42
Oleh Kozynets +
OlivierLuG

Development Team
Paolo Lammens +
Paul Ganssle
Pax +
Peter Liu +
Philip Cerles +
Pranjal Bhardwaj +
Prayag Savsani +
Purushothaman Srikanth + Qbiwan +
Rahul Chauhan +
Rahul Sathanapalli +
Rajat Bishnoi +
Ray Bell
Reshama Shaikh +
Richard Shadrach
Robert Bradshaw
Robert de Vries

-
-
-

Rohith295
S Mono +
S.TAKENO +
Sahid Velji + Sam Cohen +
Sam Ezebunandu +
Sander +
Sarthak +
Sarthak Vineet Kumar +
Satrio H Wicaksono +
Scott Lasley
Shao Yang Hong + • Sharon Woo + Shubham Mehra +

Simon Hawkins
Sixuan (Cherie) Wu +
Souris Ash +
Steffen Rehberg
Suvayu Ali
Sven
SylvainLan +
T. JEGHAM +
Terji Petersen
Thomas Dickson +
Thomas Heavey +
Thomas Smith

-
-
-

Tobias Pitters
Tom Augspurger
Tomasz Sakrejda +
Torsten Wörtwein +
Ty Mick +
UrielMaD +
Uwe L. Korn
Vikramaditya Gaonkar +
VirosaLi +
W.R +
Warren White +
Wesley Boelrijk +
William Ayd
Yanxian Lin + Yassir Karroum + Yong Kai Yi +
Yuanhao Geng +
Yury Mikhaylov +
Yutaro Ikeda
Yuya Takashina +
Zach Brookler +
Zak Kohler +

ZhihuiChen0903 +
abmyii
alexhtn +

-
-
-

asharma13524 +
attack68
beanan +
chinhwee
cleconte987
danchev +
ebardie +
edwardkong
elliott rampono +
estasney +
gabicca
geetha-rangaswamaiah + gfyoun
guru kiran
hardikpnsp +
icanhazcodeplz +
ivanovmg +
jbrockmendel
jeschwar
jnecus
jooooey +
junk +
krajatcl +
lacrosse91 +

-
-
-

leo +
lpkirwin +

lrjball

lucasrodes + ma3da +

-
-
-
-
-
-

-

mavismonica + mlondschien + mzeitlin11 + nguevara +

nrebena

parkdj1 +

partev

patrick

realead

rxsg +

samilAyoub +

sanderland

shawnbrown

sm1899 +

smartvinnetou

ssortman +

steveya +

taytzehao +

tiagohonorato +

timhunderwood

tkmz-n +

tnwei +

tpanza +

vineethraj510 +

vmdhhh +

xinrong-databricks +

yonas kassa +

yonashub +

Ádám Lippai +

5.2 Version 1.1

5.2.1 What's new in 1.1.5 (December 07, 2020)

These are the changes in 1.1.5. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in addition of a timedelta-like scalar to a *DatetimeIndex* raising incorrectly (GH37295)

Fixed regression in *Series.groupby()* raising when the *Index* of the *Series* had a tuple as its name (GH37755)

Fixed regression in *DataFrame.loc()* and *Series.loc()* for `__setitem__` when onedimensional tuple was given to select from *MultiIndex* (GH37711)

Fixed regression in inplace operations on *Series* with *ExtensionDtype* with NumPy dtyped operand (GH37910)

Fixed regression in metadata propagation for groupby iterator (GH37343)

Fixed regression in *MultiIndex* constructed from a *DatetimeIndex* not retaining frequency (GH35563)

Fixed regression in *Index* constructor raising a *AttributeError* when passed a *SparseArray* with *datetime64* values (GH35843)

Fixed regression in *DataFrame.unstack()* with columns with integer dtype (GH37115) Fixed regression in indexing on a *Series* with *CategoricalDtype* after unpickling (GH37631)

Fixed regression in *DataFrame.groupby()* aggregation with out-of-bounds datetime objects in an *objectdtype* column (GH36003)

Fixed regression in *df.groupby(..).rolling(..)* with the resulting *MultiIndex* when grouping by a label that is in the index (GH37641)

Fixed regression in *DataFrame.fillna()* not filling NaN after other operations such as *DataFrame.pivot()* (GH36495).

Fixed performance regression in *df.groupby(..).rolling(..)* (GH38038)

Fixed regression in *MultiIndex.intersection()* returning duplicates when at least one of the indexes had duplicates (GH36915)

Fixed regression in *GroupBy.first()* and *GroupBy.last()* where *None* was considered a non-NA value (GH38286)

Bug fixes

Bug in *pytables* methods in python 3.9 (GH38041)

5.2. Version 1.1

Other

Only set `-Werror` as a compiler flag in the CI jobs (GH33315, GH33314)

Contributors

A total of 12 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Andrew Wieteska

Fangchen Li

Janus

Joris Van den Bossche

Matthew Roeschke

MeeseeksMachine

Development Team

Richard Shadrach

Simon Hawkins

Uwe L. Korn

jbrockmendel

patrick

5.2.2 What’s new in 1.1.4 (October 30, 2020)

These are the changes in 1.1.4. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in `read_csv()` raising a `ValueError` when `names` was of type `dict_keys` (GH36928)

Fixed regression in `read_csv()` with more than 1M rows and specifying a `index_col` argument (GH37094)

Fixed regression where attempting to mutate a `DateOffset` object would no longer raise an `AttributeError` (GH36940)

Fixed regression where `DataFrame.agg()` would fail with `TypeError` when passed positional arguments to be passed on to the aggregation function (GH36948).

Fixed regression in `RollingGroupby` with `sort=False` not being respected (GH36889) Fixed regression in `Series.astype()` converting `None` to `"nan"`

when casting to string (GH36904)

Fixed regression in *Series.rank()* method failing for read-only data (GH37290) Fixed regression in *RollingGroupby* causing a segmentation fault with Index of dtype object (GH36727)

Fixed regression in *DataFrame.resample(...).apply(...)*() raised *AttributeError* when input was a *DataFrame* and only a *Series* was evaluated (GH36951)

Fixed regression in *DataFrame.groupby(...).std()* with nullable integer dtype (GH37415) Fixed regression in *PeriodDtype* comparing both equal and unequal to its string representation (GH37265)

Fixed regression where slicing *DatetimeIndex* raised *AssertionError* on irregular time series with *pd.NaT* or on unsorted indices (GH36953 and GH35509)

Fixed regression in certain offsets (*pd.offsets.Day()* and below) no longer being hashable (GH37267)

Fixed regression in *StataReader* which required *chunksize* to be manually set when using an iterator to read a dataset (GH37280)

Fixed regression in *setitem* with *DataFrame.iloc()* which raised error when trying to set a value while filtering with a boolean list (GH36741)

Fixed regression in *setitem* with a *Series* getting aligned before setting the values (GH37427)

Fixed regression in *MultiIndex.is_monotonic_increasing* returning wrong results with NaN in at least one of the levels (GH37220)

Fixed regression in inplace arithmetic operation on a *Series* not updating the parent *DataFrame* (GH36373)

Bug fixes

Bug causing *groupby(...).sum()* and similar to not preserve metadata (GH29442)

Bug in *Series.isin()* and *DataFrame.isin()* raising a *ValueError* when the target was read-only (GH37174)

Bug in *GroupBy.fillna()* that introduced a performance regression after 1.0.5 (GH36757) Bug in *DataFrame.info()* was raising a *KeyError* when the *DataFrame* has integer column names (GH37245)

Bug in *DataFrameGroupby.apply()* would drop a *CategoricalIndex* when

grouped on (GH35792)

Contributors

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Daniel Saxton

Fangchen Li

Janus +

Joris Van den Bossche

Kevin Sheppard

Marco Gorelli

Matt Roeschke

Matthew Roeschke

MeeseeksMachine

Development Team

Paul Ganssle

Richard Shadrach

Simon Hawkins

5.2. Version 1.1

Thomas Smith

Tobias Pitters

abmyii +

jbrockmendel

patrick

5.2.3 What’s new in 1.1.3 (October 5, 2020)

These are the changes in 1.1.3. See *Release notes* for a full changelog including other versions of .

Enhancements

Added support for new Python version

1.1.3 now supports Python 3.9 (GH36296).

Development Changes

The minimum version of Cython is now the most recent bug-fix version (0.29.21) (GH36296).

Fixed regressions

Fixed regression in *DataFrame.agg()*, *DataFrame.apply()*, *Series.agg()*, and *Series.apply()* where internal suffix is exposed to the users when no relabelling is applied (GH36189)

Fixed regression in IntegerArray unary plus and minus operations raising a *TypeError* (GH36063)

Fixed regression when adding a *timedelta_range()* to a *Timestamp* raised a *ValueError* (GH35897) • Fixed regression in *Series.__getitem__()* incorrectly raising when the input was a tuple (GH35534)

Fixed regression in *Series.__getitem__()* incorrectly raising when the input was a frozenset (GH35747)

Fixed regression in modulo of *Index*, *Series* and *DataFrame* using numexpr using C not Python semantics (GH36047, GH36526)

Fixed regression in *read_excel()* with engine="odf" caused *UnboundLocalError* in some cases where cells had nested child nodes (GH36122, GH35802)

Fixed regression in *DataFrame.replace()* inconsistent replace when using a float in the replace method (GH35376)

Fixed regression in *Series.loc()* on a *Series* with a *MultiIndex* containing *Timestamp* raising *InvalidIndexError* (GH35858)

Fixed regression in *DataFrame* and *Series* comparisons between numeric arrays and strings (GH35700, GH36377)

Fixed regression in *DataFrame.apply()* with raw=True and user-function returning string (GH35940)

Fixed regression when setting empty *DataFrame* column to a *Series* in preserving name of index in frame (GH36527)

Fixed regression in *Period* incorrect value for ordinal over the maximum timestamp (GH36430)

Fixed regression in *read_table()* raised *ValueError* when delim_whitespace was set to True (GH35958)

Fixed regression in *Series.dt.normalize()* when normalizing pre-epoch dates the result was shifted one day (GH36294)

Bug fixes

Bug in *read_spss()* where passing a *pathlib.Path* as path would raise a *TypeError* (GH33666)

Bug in *Series.str.startswith()* and *Series.str.endswith()* with category dtype

not propagating na parameter (GH36241)

Bug in *Series* constructor where integer overflow would occur for sufficiently large scalar inputs when an index was provided (GH36291)

Bug in *DataFrame.sort_values()* raising an *AttributeError* when sorting on a key that casts column to categorical dtype (GH36383)

Bug in *DataFrame.stack()* raising a *ValueError* when stacking *MultiIndex* columns based on position when the levels had duplicate names (GH36353)

Bug in *Series.astype()* showing too much precision when casting from *np.float32* to string dtype (GH36451)

Bug in *Series.isin()* and *DataFrame.isin()* when using NaN and a row length above 1,000,000 (GH22205)

Bug in *cut()* raising a *ValueError* when passed a *Series* of labels with *ordered=False* (GH36603)

Other

Reverted enhancement added in -1.1.0 where *timedelta_range()* infers a frequency when passed start, stop, and periods (GH32377)

Contributors

A total of 16 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Asish Mahapatra

Dan Moore +

Daniel Saxton

Fangchen Li

Hans

Irv Lustig

Joris Van den Bossche

Kaiqi Dong

MeeseeksMachine

Number42 +

Development Team

5.2. Version 1.1

Richard Shadrach

Simon Hawkins

jbrockmendel

nrebena

patrick

5.2.4 What's new in 1.1.2 (September 8, 2020)

These are the changes in 1.1.2. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Regression in `DatetimeIndex.intersection()` incorrectly raising `AssertionError` when intersecting against a list (GH35876)

Fix regression in updating a column inplace (e.g. using `df['col'].fillna(..., inplace=True)`) (GH35731)

Fix regression in `DataFrame.append()` mixing tz-aware and tz-naive datetime columns (GH35460)

Performance regression for `RangeIndex.format()` (GH35712)

Regression where `MultiIndex.get_loc()` would return a slice spanning the full index when passed an empty list (GH35878)

Fix regression in invalid cache after an indexing operation; this can manifest when setting which does not update the data (GH35521)

Regression in `DataFrame.replace()` where a `TypeError` would be raised when attempting to replace elements of type `Interval` (GH35931)

Fix regression in pickle roundtrip of the closed attribute of `IntervalIndex` (GH35658) Fixed regression in `DataFrameGroupBy.agg()` where a

`ValueError: buffer source array is read-only` would be raised when the underlying array is read-only (GH36014) Fixed regression in `Series.groupby.rolling()` number of levels of `MultiIndex` in input was compressed to one (GH36018)

Fixed regression in `DataFrameGroupBy` on an empty `DataFrame` (GH36197)

Bug fixes

Bug in `DataFrame.eval()` with object dtype column binary operations (GH35794)

Bug in `Series` constructor raising a `TypeError` when constructing sparse datetime64 dtypes (GH35762)

Bug in `DataFrame.apply()` with `result_type="reduce"` returning with incorrect index (GH35683)

Bug in `Series.astype()` and `DataFrame.astype()` not respecting the errors

argument when set to "ignore" for extension dtypes (GH35471)

Bug in `DateTimeIndex.format()` and `PeriodIndex.format()` with `name=True` setting the first item to "None" where it should be "" (GH35712)

Bug in `Float64Index.__contains__()` incorrectly raising `TypeError` instead of returning `False` (GH35788)

Bug in `Series` constructor incorrectly raising a `TypeError` when passed an ordered set (GH36044)

Bug in `Series.dt.isocalendar()` and `DatetimeIndex.isocalendar()` that returned incorrect year for certain dates (GH36032)

Bug in `DataFrame` indexing returning an incorrect `Series` in some cases when the series has been altered and a cache not invalidated (GH33675)

Bug in `DataFrame.corr()` causing subsequent indexing lookups to be incorrect (GH35882)

Bug in `import_optional_dependency()` returning incorrect package names in cases where package name is different from import name (GH35948)

Bug when setting empty `DataFrame` column to a `Series` in preserving name of index in frame (GH31368)

Other

`factorize()` now supports `na_sentinel=None` to include NaN in the uniques of the values and remove `dropna` keyword which was unintentionally exposed to public facing API in 1.1 version from `factorize()` (GH35667)

`DataFrame.plot()` and `Series.plot()` raise `UserWarning` about usage of `FixedFormatter` and

`FixedLocator` (GH35684 and GH35945)

Contributors

A total of 16 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Ali McMaster

Asish Mahapatra

Daniel Saxton

Fangchen Li

Harsh Sharma +
Irv Lustig
Jeet Parekh +
Joris Van den Bossche
Kaiqi Dong
Matthew Roeschke MeeseeksMachine

Development Team Simon Hawkins
Terji Petersen
jbrockmendel
patrick

5.2. Version 1.1

5.2.5 What's new in 1.1.1 (August 20, 2020)

These are the changes in 1.1.1. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in `CategoricalIndex.format()` where, when stringified scalars had different lengths, the shorter string would be right-filled with spaces, so it had the same length as the longest string (GH35439)

Fixed regression in `Series.truncate()` when trying to truncate a single-element series (GH35544)

Fixed regression where `DataFrame.to_numpy()` would raise a `RuntimeError` for mixed dtypes when converting to str (GH35455)

Fixed regression where `read_csv()` would raise a `ValueError` when `.options.mode.use_inf_as_na` was set to `True` (GH35493)

Fixed regression where `.testing.assert_series_equal()` would raise an error when nonnumeric dtypes were passed with `check_exact=True` (GH35446)

Fixed regression in `.groupby(..).rolling(..)` where column selection was ignored (GH35486)

Fixed regression where `DataFrame.interpolate()` would raise a `TypeError` when the `DataFrame` was empty (GH35598)

Fixed regression in `DataFrame.shift()` with `axis=1` and heterogeneous dtypes (GH35488)

Fixed regression in `DataFrame.diff()` with read-only data (GH35559)

Fixed regression in `.groupby(..).rolling(..)` where a segfault would occur with

center=True and an odd number of values (GH35552)

Fixed regression in *DataFrame.apply()* where functions that altered the input in-place only operated on a single row (GH35462)

Fixed regression in *DataFrame.reset_index()* would raise a *ValueError* on empty *DataFrame* with a *MultiIndex* with a *datetime64* dtype level (GH35606, GH35657) Fixed regression where *.merge_asof()* would raise a *UnboundLocalError* when *left_index*, *right_index* and *tolerance* were set (GH35558)

Fixed regression in *.groupby(..).rolling(..)* where a custom *BaseIndexer* would be ignored (GH35557)

Fixed regression in *DataFrame.replace()* and *Series.replace()* where compiled regular expressions would be ignored during replacement (GH35680)

Fixed regression in *aggregate()* where a list of functions would produce the wrong results if at least one of the functions did not aggregate (GH35490)

Fixed memory usage issue when instantiating large *.arrays.StringArray* (GH35499) **Bug fixes**

Bug in *Styler* whereby *cell_ids* argument had no effect due to other recent changes (GH35588) (GH35663)

Bug in *.testing.assert_series_equal()* and *.testing.assert_frame_equal()* where extension dtypes were not ignored when *check_dtypes* was set to *False* (GH35715)

Bug in *to_timedelta()* fails when *arg* is a *Series* with *Int64* dtype containing null values (GH35574)

Bug in *.groupby(..).rolling(..)* where passing closed with column selection would raise a *ValueError* (GH35549)

Bug in *DataFrame* constructor failing to raise *ValueError* in some cases when data and index have mismatched lengths (GH33437)

Contributors

A total of 20 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Ali McMaster

Daniel Saxton
Eric Goddard +
Fangchen Li
Isaac Virshup
Joris Van den Bossche
Kevin Sheppard
Matthew Roeschke
MeeseeksMachine +

Development Team
Richard Shadrach
Simon Hawkins
Terji Petersen
Tom Augspurger
Yutaro Ikeda +
attack68 +
edwardkong +
gabicca +
jbrockmendel
sanderland +

5.2. Version 1.1

5.2.6 What's new in 1.1.0 (July 28, 2020)

These are the changes in 1.1.0. See *Release notes* for a full changelog including other versions of .

Enhancements

KeyErrors raised by loc specify missing labels

Previously, if labels were missing for a `.loc` call, a `KeyError` was raised stating that this was no longer supported.

Now the error message also includes a list of the missing labels (max 10 items, display width 80 characters). See GH34272.

All dtypes can now be converted to `StringDtype`

Previously, declaring or converting to `StringDtype` was in general only possible if the data was already only str or nan-like (GH31204). `StringDtype` now works in all situations where `astype(str)` or `dtype=str` work:

For example, the below now works:

In [1]: ser = pd.Series([1, "abc", np.nan], dtype="string")

In [2]: ser **Out[2]:**

1

abc

<NA>

Length: 3, dtype: string

In [3]: ser[0]

Out[3]: '1'

In [4]: pd.Series([1, 2, np.nan], dtype="Int64").astype("string") **Out[4]:**

1

2

<NA>

Length: 3, dtype: string

Non-monotonic PeriodIndex partial string slicing

PeriodIndex now supports partial string slicing for non-monotonic indexes, mirroring *DatetimeIndex* behavior (GH31096)

For example:

In [5]: dti = pd.date_range("2014-01-01", periods=30, freq="30D")

In [6]: pi = dti.to_period("D")

In [7]: ser_monotonic = pd.Series(np.arange(30), index=pi)

In [8]: shuffler = list(range(0, 30, 2)) + list(range(1, 31, 2))

In [9]: ser = ser_monotonic[shuffler] **In [10]:** ser **Out[10]:**

2014-01-01 0

2014-03-02 2

2014-05-01 4

2014-06-30 6

2014-08-29 8

..

2015-09-23 21

2015-11-22 23

2016-01-21 25

2016-03-21 27

2016-05-20 29

Freq: D, Length: 30, dtype: int64 **In [11]:** ser["2014"] **Out[11]:**

```

2014-01-01 0
2014-03-02 2
2014-05-01 4
2014-06-30 6
2014-08-29 8
2014-10-28 10
2014-12-27 12
2014-01-31 1
2014-04-01 3
2014-05-31 5
2014-07-30 7
2014-09-28 9
2014-11-27 11
Freq: D, Length: 13, dtype: int64 In [12]: ser.loc["May 2015"] Out[12]:
2015-05-26 17
Freq: D, Length: 1, dtype: int64

```

Comparing two DataFrame or two Series and summarizing the differences We've added *DataFrame.compare()* and *Series.compare()* for comparing two DataFrame or two Series (GH30429)

```

In [13]: df = pd.DataFrame(
....: {
....: "col1": ["a", "a", "b", "b", "a"],
....: "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
....: "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
....: },
....: columns=["col1", "col2", "col3"],

....: ) ....:

```

```

In [14]: df Out[14]:

```

```

col1 col2 col3
a 1.0 1.0
a 2.0 2.0
b 3.0 3.0
b NaN 4.0
a 5.0 5.0 [5 rows x 3 columns]

```

In [15]: df2 = df.copy()

In [16]: df2.loc[0, 'col1'] = 'c' **In [17]:** df2.loc[2, 'col3'] = 4.0

In [18]: df2 **Out[18]:**

```
col1 col2 col3
```

```
c 1.0 1.0
```

```
a 2.0 2.0
```

```
b 3.0 4.0
```

```
b NaN 4.0
```

```
a 5.0 5.0
```

```
[5 rows x 3 columns]
```

In [19]: df.compare(df2) **Out[19]:**

```
col1 col3
```

```
self other self other
```

```
0 a c NaN NaN
```

```
2 NaN NaN 3.0 4.0
```

```
[2 rows x 4 columns]
```

See *User Guide* for more details.

Allow NA in groupby key

With *groupby*, we've added a *dropna* keyword to *DataFrame.groupby()* and *Series.groupby()* in order to allow NA values in group keys. Users can define *dropna* to *False* if they want to include NA values in groupby keys. The default is set to *True* for *dropna* to keep backwards compatibility (GH3729)

In [20]: df_list = [[1, 2, 3], [1, **None**, 4], [2, 1, 3], [1, 2, 2]]

In [21]: df_dropna = pd.DataFrame(df_list, columns=["a", "b", "c"])

In [22]: df_dropna **Out[22]:**

```
a b c
```

```
1 2.0 3
```

```
1 NaN 4
```

```
2 1.0 3
```

```
1 2.0 2
```

```
[4 rows x 3 columns]
```

Default ``dropna`` is set to *True*, which will exclude NaNs in keys **In [23]:**

df_dropna.groupby(by=["b"], dropna=**True**).sum() **Out[23]:**


```
a c
b
1.0 2 3
2.0 2 5
rows x 2 columns]
```

In order to allow NaN in keys, set ``dropna`` to False **In [24]:**
df_dropna.groupby(by=["b"], dropna=False).sum() **Out[24]:**

```
a c
b
1.0 2 3
2.0 2 5
NaN 1 4
rows x 2 columns]
```

The default setting of dropna argument is True which means NA are not included in group keys.

Sorting with keys

We've added a key argument to the *DataFrame* and *Series* sorting methods, including *DataFrame.sort_values()*, *DataFrame.sort_index()*,

Series.sort_values() , and *Series.sort_index()*. The key can be any callable function which is applied column-by-column to each column used for sorting, before sorting is performed (GH27237). See *sort_values with keys* and *sort_index with keys* for more information.

In [25]: s = pd.Series(['C', 'a', 'B'])

In [26]: s **Out[26]:**

```
C
a
B
Length: 3, dtype: object
```

In [27]: s.sort_values() **Out[27]:**

```
2 B
C
a
Length: 3, dtype: object
```

Note how this is sorted with capital letters first. If we apply the *Series.str.lower()* method, we get

In [28]: s.sort_values(key=**lambda** x: x.str.lower()) **Out[28]:**

```
a
B
0 C
Length: 3, dtype: object
```

When applied to a DataFrame, the key is applied per-column to all columns or a subset if *by* is specified, e.g.

In [29]: df = pd.DataFrame({'a': ['C', 'C', 'a', 'a', 'B', 'B'],
....: 'b': [1, 2, 3, 4, 5, 6]})

....:

In [30]: df **Out[30]:** a b 0 C 1

```
C 2
a 3
a 4
B 5
B 6
[6 rows x 2 columns]
```

In [31]: df.sort_values(by=['a'], key=**lambda** col: col.str.lower()) **Out[31]:** a

```
b 2 a 3
```

```
a 4
B 5
B 6
C 1
C 2
```

```
[6 rows x 2 columns]
```

For more details, see examples and documentation in *DataFrame.sort_values()*, *Series.sort_values()*, and *sort_index()*.

Fold argument support in Timestamp constructor

Timestamp: now supports the keyword-only *fold* argument according to PEP 495 similar to parent *datetime.datetime* class. It supports both accepting *fold* as an initialization argument and inferring *fold* from other constructor

arguments (GH25057, GH31338). Support is limited to dateutil timezones as pytz doesn't support fold.

For example:

In [32]: ts = pd.Timestamp("2019-10-27 01:30:00+00:00")

In [33]: ts.fold

Out[33]: 0

In [34]: ts = pd.Timestamp(year=2019, month=10, day=27, hour=1, minute=30,

....: tz="dateutil/Europe/London", fold=1)

(continues on next page)

(continued from previous page)

....:

In [35]: ts

Out[35]: Timestamp('2019-10-27 01:30:00+0000',
tz='dateutil//usr/share/zoneinfo/ ,_ Europe/London')

For more on working with fold, see *Fold subsection* in the user guide.

Parsing timezone-aware format with different timezones in `to_datetime`

`to_datetime()` now supports parsing formats containing timezone names (%Z) and UTC offsets (%z) from different timezones then converting them to UTC by setting `utc=True`. This would return a *DatetimeIndex* with timezone at UTC as opposed to an *Index* with object dtype if `utc=True` is not set (GH32792).

For example:

In [36]: tz_strs = ["2010-01-01 12:00:00 +0100", "2010-01-01 12:00:00 -0100",

....: "2010-01-01 12:00:00 +0300", "2010-01-01 12:00:00 +0400"]:

In [37]: pd.to_datetime(tz_strs, format='%Y-%m%d %H:%M:%S %z',
utc=True) **Out[37]:**

Out[37]: 01 09:00:00+00:00', '2010-01-01 08:00:00+00:00'],
dtype='datetime64[ns, UTC]', freq=None)

In [38]: `pd.to_datetime(tz_strs, format='%Y-%m%d %H:%M:%S %z')`
Out[38]: `Index([2010-01-01 12:00:00+01:00, 2010-01-01 12:00:00-01:00, 2010-01-01 12:00:00+03:00, 2010-01-01 12:00:00+04:00], dtype='object')`

Grouper and resample now supports the arguments origin and offset
Grouper and *DataFrame.resample()* now supports the arguments *origin* and *offset*. It let the user control the timestamp on which to adjust the grouping. (GH31809)

The bins of the grouping are adjusted based on the beginning of the day of the time series starting point. This works well with frequencies that are multiples of a day (like 30D) or that divides a day (like 90s or 1min). But it can create inconsistencies with some frequencies that do not meet this criteria. To change this behavior you can now specify a fixed timestamp with the argument *origin*.

Two arguments are now deprecated (more information in the documentation of *DataFrame.resample()*):

base should be replaced by *offset*.

loffset should be replaced by directly adding an offset to the index *DataFrame* after being resampled.

Small example of the use of *origin*:

In [39]: `start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'`

In [40]: `middle = '2000-10-02 00:00:00'`

In [41]: `rng = pd.date_range(start, end, freq='7min')`

(continues on next page)

(continued from previous page)

In [42]: `ts = pd.Series(np.arange(len(rng)) * 3, index=rng)`

In [43]: `ts` **Out[43]:**

2000-10-01 23:30:00 0

2000-10-01 23:37:00 3

2000-10-01 23:44:00 6

2000-10-01 23:51:00 9

2000-10-01 23:58:00 12

2000-10-02 00:05:00 15

2000-10-02 00:12:00 18

2000-10-02 00:19:00 21

```
2000-10-02 00:26:00 24
Freq: 7T, Length: 9, dtype: int64
```

Resample with the default behavior 'start_day' (origin is 2000-10-01 00:00:00): **In [44]:** ts.resample('17min').sum() **Out[44]:**

```
2000-10-01 23:14:00 0
2000-10-01 23:31:00 9
2000-10-01 23:48:00 21
2000-10-02 00:05:00 54
2000-10-02 00:22:00 24
Freq: 17T, Length: 5, dtype: int64
```

In [45]: ts.resample('17min', origin='start_day').sum() **Out[45]:**

```
2000-10-01 23:14:00 0
2000-10-01 23:31:00 9
2000-10-01 23:48:00 21
2000-10-02 00:05:00 54
2000-10-02 00:22:00 24
Freq: 17T, Length: 5, dtype: int64
```

Resample using a fixed origin:

In [46]: ts.resample('17min', origin='epoch').sum() **Out[46]:**

```
2000-10-01 23:18:00 0
2000-10-01 23:35:00 18
2000-10-01 23:52:00 27
2000-10-02 00:09:00 39
2000-10-02 00:26:00 24
Freq: 17T, Length: 5, dtype: int64
```

In [47]: ts.resample('17min', origin='2000-01-01').sum() **Out[47]:**

```
2000-10-01 23:24:00 3
2000-10-01 23:41:00 15
2000-10-01 23:58:00 45
2000-10-02 00:15:00 45
Freq: 17T, Length: 4, dtype: int64
```

If needed you can adjust the bins with the argument *offset* (a *Timedelta*) that would be added to the default origin.

For a full example, see: *Use origin or offset to adjust the start of the bins.*

fsspec now used for filesystem handling

For reading and writing to filesystems other than local and reading from HTTP(S), the optional dependency fsspec will be used to dispatch operations (GH33452). This will give unchanged functionality for S3 and GCS storage, which were already supported, but also add support for several other storage implementations such as Azure Datalake and Blob, SSH, FTP, dropbox and github. For docs and capabilities, see the fsspec docs.

The existing capability to interface with S3 and GCS will be unaffected by this change, as fsspec will still bring in the same packages as before.

Other enhancements

Compatibility with matplotlib 3.3.0 (GH34850)

IntegerArray.astype() now supports datetime64 dtype (GH32538)

IntegerArray now implements the sum operation (GH33172)

Added *.errors.InvalidIndexError* (GH34570).

Added *DataFrame.value_counts()* (GH5377)

Added a *.api.indexers.FixedForwardWindowIndexer()* class to support forwardlooking windows during rolling operations.

Added a *.api.indexers.VariableOffsetWindowIndexer()* class to support rolling operations with non-fixed offsets (GH34994)

describe() now includes a *datetime_is_numeric* keyword to control how datetime columns are summarized (GH30164, GH34798)

Styler may now render CSS more efficiently where multiple cells have the same styling (GH30876)

highlight_null() now accepts subset argument (GH31345)

When writing directly to a sqlite connection *DataFrame.to_sql()* now supports the multi method (GH29921)

.errors.OptionError is now exposed in *.errors* (GH27553)

Added *api.extensions.ExtensionArray.argmax()* and *api.extensions.ExtensionArray.argmin()* (GH24382)

timedelta_range() will now infer a frequency when passed start, stop, and periods (GH32377)

Positional slicing on a *IntervalIndex* now supports slices with step > 1 (GH31658) *Series.str* now has a *fullmatch* method that matches a regular expression against the entire string in each row of the *Series*, similar to *re.fullmatch* (GH32806).

DataFrame.sample() will now also allow array-like and *BitGenerator* objects to be passed to *random_state* as seeds (GH32503)

Index.union() will now raise *RuntimeWarning* for *MultiIndex* objects if the object inside are unsortable. Pass *sort=False* to suppress this warning (GH33015)

Added *Series.dt.isocalendar()* and *DatetimeIndex.isocalendar()* that returns a *DataFrame* with year, week, and day calculated according to the ISO 8601 calendar (GH33206, GH34392).

5.2. Version 1.1

The *DataFrame.to_feather()* method now supports additional keyword arguments (e.g. to set the compression) that are added in *pyarrow* 0.17 (GH33422).

The *cut()* will now accept parameter *ordered* with default *ordered=True*. If *ordered=False* and no labels are provided, an error will be raised (GH33141)

DataFrame.to_csv(), *DataFrame.to_pickle()*, and *DataFrame.to_json()* now support passing a dict of compression arguments when using the *gzip* and *bz2* protocols. This can be used to set a custom compression level, e.g.,
df.to_csv(path,
compression={'method': 'gzip', 'compresslevel': 1}) (GH33196)

melt() has gained an *ignore_index* (default *True*) argument that, if set to *False*, prevents the method from dropping the index (GH17440).

Series.update() now accepts objects that can be coerced to a *Series*, such as dict and list, mirroring the behavior of *DataFrame.update()* (GH33215)

transform() and *aggregate()* have gained *engine* and *engine_kwargs* arguments that support executing functions with *Numba* (GH32854, GH33388)

interpolate() now supports *SciPy* interpolation method *scipy.interpolate.CubicSpline* as method *cubic spline* (GH33670)

DataFrame.groupby and *Series.groupby* now implement the *sample* method for doing random sampling within groups (GH31775)

DataFrame.to_numpy() now supports the *na_value* keyword to control the NA sentinel in the output array (GH33820)

Added *api.extension.ExtensionArray.equals* to the extension array interface, similar to *Series.equals()* (GH27081)

The minimum supported dta version has increased to 105 in *read_stata()* and *StataReader* (GH26667).

to_stata() supports compression using the *compression* keyword argument. Compression can either be inferred or explicitly set using a string or a dictionary containing both the method and any additional arguments that are passed to the compression library. Compression was also added to the low-level Stata-file writers *StataWriter*, *StataWriter117*, and *StataWriterUTF8* (GH26599).

HDFStore.put() now accepts a *track_times* parameter. This parameter is passed to the *create_table* method of *PyTables* (GH32682).

Series.plot() and *DataFrame.plot()* now accepts *xlabel* and *ylabel* parameters to present labels on x and y axis (GH9093).

Made *.core.window.rolling.Rolling* and *.core.window.expanding.Expanding* iterable (GH11704)

Made *option_context* a *contextlib.ContextDecorator*, which allows it to be used as a decorator over an entire function (GH34253).

DataFrame.to_csv() and *Series.to_csv()* now accept an *errors* argument (GH22610) *transform()* now allows *func* to be *pad*, *backfill* and *cumcount* (GH31269). *read_json()* now accepts an *nrows* parameter. (GH33916).

DataFrame.hist() , *Series.hist()*, *core.groupby.DataFrameGroupBy.hist()*, and *core.groupby.SeriesGroupBy.hist()* have gained the *legend* argument. Set to *True* to show a legend in the histogram. (GH6279)

concat() and *append()* now preserve extension dtypes, for example combining a nullable integer column with a numpy integer column will no longer result in object dtype but preserve the integer dtype (GH33607, GH34339, GH34095).

read_gbq() now allows to disable progress bar (GH33360).

read_gbq() now supports the *max_results* kwarg from *-gbq* (GH34639).

DataFrame.cov() and *Series.cov()* now support a new parameter *ddof* to support delta degrees of freedom as in the corresponding numpy methods (GH34611). *DataFrame.to_html()* and *DataFrame.to_string()*'s *col_space* parameter now accepts a list or dict to change only some specific columns' width (GH28917).

DataFrame.to_excel() can now also write OpenOffice spreadsheet (.ods) files (GH27222)

explode() now accepts *ignore_index* to reset the index, similar to *pd.concat()* or *DataFrame.sort_values()* (GH34932).

DataFrame.to_markdown() and *Series.to_markdown()* now accept *index* argument as an alias for *tabulate*'s *showindex* (GH32667)

read_csv() now accepts string values like "0", "0.0", "1", "1.0" as convertible to the nullable Boolean dtype (GH34859)

.core.window.ExponentialMovingWindow now supports a *times* argument that allows mean to be calculated with observations spaced by the timestamps in times (GH34839) *DataFrame.agg()* and *Series.agg()* now accept named aggregation for renaming the output columns/indexes. (GH26513)
compute.use_numba now exists as a configuration option that utilizes the numba engine when available (GH33966, GH35374)

Series.plot() now supports asymmetric error bars. Previously, if *Series.plot()* received a "2xN" array with error values for *yerr* and/or *xerr*, the left/lower values (first row) were mirrored, while the right/upper values (second row) were ignored. Now, the first row represents the left/lower error values and the second row the right/upper error values. (GH9536)

Notable bug fixes

These are bug fixes that might have notable behavior changes.

MultiIndex.get_indexer interprets method argument correctly

This restores the behavior of *MultiIndex.get_indexer()* with *method='backfill'* or *method='pad'* to the behavior before 0.23.0. In particular, *MultiIndexes* are treated as a list of tuples and padding or backfilling is done with respect to the ordering of these lists of tuples (GH29896).

As an example of this, given:

```
In [48]: df = pd.DataFrame({
```

```
....: 'a': [0, 0, 0, 0],
```

```
....: 'b': [0, 2, 3, 4],
```

```
....: 'c': ['A', 'B', 'C', 'D'],
```

```
....: }).set_index(['a', 'b']) ....:
```

```
In [49]: mi_2 = pd.MultiIndex.from_product([[0], [-1, 0, 1, 3, 4, 5]])
```

The differences in reindexing df with mi_2 and using method='backfill' can be seen here:

>= 0.23, < 1.1.0: 5.2. Version 1.1

```
In [1]: df.reindex(mi_2, method='backfill') Out[1]: c
```

```
0 -1 A
```

```
A
```

```
D
```

```
A
```

```
A
```

```
C
```

<0.23, >= 1.1.0

```
In [50]: df.reindex(mi_2, method='backfill') Out[50]:
```

```
c
```

```
0 -1 A
```

```
A
```

```
B
```

```
C
```

```
D
```

```
NaN
```

```
[6 rows x 1 columns]
```

And the differences in reindexing df with mi_2 and using method='pad' can be seen here: *>= 0.23, < 1.1.0*

```
In [1]: df.reindex(mi_2, method='pad') Out[1]:
```

```
c
```

```
0 -1 NaN
```

```
NaN
```

D
NaN
A
C

$< 0.23, \geq 1.1.0$

In [51]: df.reindex(mi_2, method='pad') **Out[51]:**

c
0 -1 NaN
A
A
C
D
D
[6 rows x 1 columns]

Failed label-based lookups always raise KeyError

Label lookups `series[key]`, `series.loc[key]` and `frame.loc[key]` used to raise either `KeyError` or

`TypeError` depending on the type of key and type of *Index*. These now consistently raise `KeyError` (GH31867)

In [52]: ser1 = pd.Series(range(3), index=[0, 1, 2]) **In [53]:** ser2 = pd.Series(range(3), index=pd.date_range("2020-02-01", periods=3))

Previous behavior:

In [3]: ser1[1.5] ...

TypeError: cannot do label indexing on Int64Index with these indexers [1.5] of type , → float

In [4]: ser1["foo"] ...

KeyError: 'foo'

In [5]: ser1.loc[1.5] ...

TypeError: cannot do label indexing on Int64Index with these indexers [1.5] of type , → float

In [6]: ser1.loc["foo"] ...

KeyError: 'foo'

In [7]: ser2.loc[1] ...

TypeError: cannot do label indexing on DatetimeIndex with these indexers

```
[1] of type , → int
In [8]: ser2.loc[pd.Timestamp(0)] ...
KeyError: Timestamp('1970-01-01 00:00:00')
```

New behavior : **In [3]:** ser1[1.5] ...

```
KeyError: 1.5
```

In [4] ser1["foo"] ...

```
KeyError: 'foo'
```

In [5]: ser1.loc[1.5] ...

```
KeyError: 1.5
```

In [6]: ser1.loc["foo"] ...

```
KeyError: 'foo'
```

In [7]: ser2.loc[1]

(continues on next page)

5.2. Version 1.1

(continued from previous page) ...

```
KeyError: 1
```

In [8]: ser2.loc[pd.Timestamp(0)] ...

```
KeyError: Timestamp('1970-01-01 00:00:00')
```

Similarly, *DataFrame.at()* and *Series.at()* will raise a *TypeError* instead of a *ValueError* if an incompatible key is passed, and *KeyError* if a missing key is passed, matching the behavior of *.loc[]* (GH31722)

Failed Integer Lookups on MultiIndex Raise KeyError

Indexing with integers with a *MultiIndex* that has an integer-dtype first level incorrectly failed to raise *KeyError* when one or more of those integer keys is not present in the first level of the index (GH33539)

In [54]: idx = pd.Index(range(4))

In [55]: dti = pd.date_range("2000-01-03", periods=3)

In [56]: mi = pd.MultiIndex.from_product([idx, dti]) **In [57]:** ser =
pd.Series(range(len(mi)), index=mi)

Previous behavior :

In [5]: ser[[5]] **Out[5]:** Series([], dtype: int64)

New behavior:

In [5]: ser[[5]] ...

KeyError: '[5] not in index'

DataFrame.merge() preserves right frame's row order

DataFrame.merge() now preserves the right frame's row order when executing a right merge (GH27453)

In [58]: left_df = pd.DataFrame({'animal': ['dog', 'pig'],

....: 'max_speed': [40, 11]})

....:

In [59]: right_df = pd.DataFrame({'animal': ['quetzal', 'pig'],

....: 'max_speed': [80, 11]})

....:

In [60]: left_df **Out[60]:**

animal max_speed dog 40

pig 11

[2 rows x 2 columns]

(continues on next page)

(continued from previous page)

In [61]: right_df **Out[61]:**

animal max_speed

quetzal 80

pig 11

[2 rows x 2 columns]

Previous behavior :

>>> left_df.merge(right_df, on=['animal', 'max_speed'], how="right") animal

max_speed pig 11

quetzal 80

New behavior:

In [62]: left_df.merge(right_df, on=['animal', 'max_speed'], how="right")

Out[62]: animal max_speed

quetzal 80

pig 11

[2 rows x 2 columns]

Assignment to multiple columns of a DataFrame when some columns do not exist

Assignment to multiple columns of a *DataFrame* when some of the columns do not exist would previously assign the values to the last column. Now, new columns will be constructed with the right values. (GH13658)

```
In [63]: df = pd.DataFrame({'a': [0, 1, 2], 'b': [3, 4, 5]})
```

```
In [64]: df Out[64]: a b 0 0 3
```

```
1 4
```

```
2 5
```

```
[3 rows x 2 columns]
```

Previous behavior:

```
In [3]: df[['a', 'c']] = 1
```

```
In [4]: df
```

```
Out[4]: a b 0 1 1
```

```
1 1
```

```
1 1
```

New behavior:

5.2. Version 1.1

```
In [65]: df[['a', 'c']] = 1
```

```
In [66]: df Out[66]:
```

```
a b c 0 1 3 1
```

```
1 4 1
```

```
1 5 1
```

```
[3 rows x 3 columns]
```

Consistency across groupby reductions

Using *DataFrame.groupby()* with `as_index=True` and the aggregation `nunique` would include the grouping column(s) in the columns of the result. Now the grouping column(s) only appear in the index, consistent with other reductions. (GH32579)

```
In [67]: df = pd.DataFrame({"a": ["x", "x", "y", "y"], "b": [1, 1, 2, 3]})
```

```
In [68]: df Out[68]: a b 0 x 1
```

```
x 1
```

```
y 2
```

```
y 3
```

[4 rows x 2 columns]

Previous behavior: **In [3]:** df.groupby("a", as_index=**True**).nunique()

Out[4]: a b

a
x 1 1 y 1 2

New behavior:

In [69]: df.groupby("a", as_index=**True**).nunique() **Out[69]:**

b

a
x 1 y 2

[2 rows x 1 columns]

Using *DataFrame.groupby()* with as_index=False and the function idxmax, idxmin, mad, nunique, sem, skew, or std would modify the grouping column. Now the grouping column remains unchanged, consistent with other reductions. (GH21090, GH10355)

Previous behavior :

In [3]: df.groupby("a", as_index=**False**).nunique() **Out[4]:** a b 0 1 1

1 1 2

New behavior:

In [70]: df.groupby("a", as_index=**False**).nunique() **Out[70]:** a b 0 x 1

1 y 2

[2 rows x 2 columns]

The method *size()* would previously ignore as_index=False. Now the grouping columns are returned as columns, making the result a *DataFrame* instead of a *Series*. (GH32599) *Previous behavior:*

In [3]: df.groupby("a", as_index=**False**).size() **Out[4]:** a

x 2 y 2

dtype: int64

In [71]: df.groupby("a", as_index=**False**).size() **Out[71]:**

a size

x 2

y 2

[2 rows x 2 columns]

agg() lost results with as_index=False when relabeling columns

Previously agg() lost the result columns, when the as_index option was set to False and the result columns were relabeled. In this case the result values were replaced with the previous index (GH32240).

```
In [72]: df = pd.DataFrame({"key": ["x", "y", "z", "x", "y", "z"],  
....: "val": [1.0, 0.8, 2.0, 3.0, 3.6, 0.75]})
```

```
....:
```

```
In [73]: df Out[73]:
```

```
key val 0 x 1.00
```

```
y 0.80
```

```
z 2.00
```

```
x 3.00
```

```
y 3.60
```

```
z 0.75
```

(continues on next page)

5.2. Version 1.1

(continued from previous page)

[6 rows x 2 columns]

Previous behavior:

```
In [2]: grouped = df.groupby("key", as_index=False)
```

```
In [3]: result = grouped.agg(min_val=pd.NamedAgg(column="val",  
aggfunc="min"))
```

```
In [4]: result Out[4]: min_val
```

```
x
```

```
y
```

```
z
```

New behavior:

```
In [74]: grouped = df.groupby("key", as_index=False)
```

```
In [75]: result = grouped.agg(min_val=pd.NamedAgg(column="val",  
aggfunc="min")) In [76]: result Out[76]:
```

```
key min_val
```

```
x 1.00
```

```
y 0.80
```

```
z 0.75
```


[3 rows x 2 columns]

apply and applymap on DataFrame evaluates first row/column only once

In [77]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 6]})

In [78]: def func(row):

....: print(row)

....: return row

....:

Previous behavior:

In [4]: df.apply(func, axis=1) a

1 b 3

Name: 0, dtype: int64

a 1 b 3

Name: 0, dtype: int64

a 2 b 6

Name: 1, dtype: int64 **Out[4]:**

a b 0 1 3

1 2 6

In [79]: df.apply(func, axis=1)

a 1 b 3

Name: 0, Length: 2, dtype: int64

a 2 b 6

Name: 1, Length: 2, dtype: int64

Out[79]: a b 0 1 3

1 2 6

[2 rows x 2 columns]

Backwards incompatible API changes

Added check_freq argument assert_series_equal

The check_freq argument was *assert_series_equal()* in 1.1.0 and defaults to True. *testing.assert_frame_equal()* and *testing.assert_series_equal()* now raise AssertionError if the indexes do not have the same frequency. Before

1.1.0, the index frequency was not checked.

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH33718, GH29766, GH29723, pytables $\geq 3.4.3$). If installed, we now require: to `testing.assert_frame_equal` and `testing.added to testing.assert_frame_equal() and testing.`

Package Minimum Version Required Changed

numpy 1.15.4 X X pytz 2015.4 X pythondatetime 2.7.3

bottleneck 1.2.1 numexpr 2.6.2 pytest (dev)

4.0.2 X X

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of . Optional libraries below the lowest tested version may still work, but are not considered supported.

5.2. Version 1.1 Package Minimum Changed Version

beautifulsoup4 4.6.0 fastparquet 0.3.2 fsspec 0.7.4 gcsfs 0.6.0 X lxml 3.8.0

matplotlib 2.2.2 numba 0.46.0 openpyxl 2.5.7 pyarrow 0.13.0 pymysql 0.7.1

pytables 3.4.3 X s3fs 0.4.0 X scipy 1.2.0 X sqlalchemy 1.1.4 xarray 0.8.2 xlrd

1.1.0 xlsxwriter 0.9.8 xlwt 1.2.0

-gbq 1.2.0 X

See *Dependencies* and *Optional dependencies* for more.

Development changes

The minimum version of Cython is now the most recent bug-fix version (0.29.16) (GH33334).

Deprecations

Lookups on a *Series* with a single-item list containing a slice (e.g. `ser[[slice(0, 4)]]`) are deprecated and will raise in a future version. Either convert the list to a tuple, or pass the slice directly instead (GH31333)

`DataFrame.mean()` and `DataFrame.median()` with `numeric_only=None` will include `datetime64` and `datetime64tz` columns in a future version (GH29941)
Setting values with `.loc` using a positional slice is deprecated and will raise in

a future version. Use `.loc` with labels or `.iloc` with positions instead (GH31840)

`DataFrame.to_dict()` has deprecated accepting short names for orient and will raise in a future version (GH32515)

`Categorical.to_dense()` is deprecated and will be removed in a future version, use `np.asarray(cat)` instead (GH32639)

The fastpath keyword in the `SingleBlockManager` constructor is deprecated and will be removed in a future version (GH33092)

Providing suffixes as a set in `.merge()` is deprecated. Provide a tuple instead (GH33740, GH34741).

Indexing a `Series` with a multi-dimensional indexer like `[:, None]` to return an ndarray now raises a `FutureWarning`. Convert to a NumPy array before indexing instead (GH27837)

`Index.is_mixed()` is version, deprecated and will be removed in a future check index.

`inferred_type` directly instead (GH32922) Passing any arguments but the first one to `read_html()` as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573).

Passing any arguments but `path_or_buf` (the first one) to `read_json()` as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573).

Passing any arguments but the first two to `read_excel()` as positional arguments is deprecated. All other arguments should be given as keyword arguments (GH27573). `.api.types.is_categorical()` is deprecated and will be removed in a future version; use `.api.types.is_categorical_dtype()` instead (GH33385)

`Index.get_value()` is deprecated and will be removed in a future version (GH19728) `Series.dt.week()` and `Series.dt.weekofyear()` are deprecated and will be removed in a future version, use `Series.dt.isocalendar().week()` instead (GH33595)

`DatetimeIndex.week()` and `DatetimeIndex.weekofyear` are deprecated and will be removed in a future version, use `DatetimeIndex.isocalendar().week` instead (GH33595) `DatetimeArray.week()` and `DatetimeArray.weekofyear` are

deprecated and will be removed in a future version, use `DatetimeArray.isocalendar().week` instead (GH33595) `DateOffset.__call__()` is deprecated and will be removed in a future version, use `offset + other` instead (GH34171)

`apply_index()` is deprecated and will be removed in a future version. Use `offset + other` instead (GH34580)

`DataFrame.tshift()` and `Series.tshift()` are deprecated and will be removed in a future version, use `DataFrame.shift()` and `Series.shift()` instead (GH11631)

Indexing an *Index* object with a float key is deprecated, and will raise an `IndexError` in the future. You can manually convert to an integer key instead (GH34191). The `squeeze` keyword in `groupby()` is deprecated and will be removed in a future version (GH32380)

The `tz` keyword in `Period.to_timestamp()` is deprecated and will be removed in a future version; use `per.to_timestamp(...).tz_localize(tz)` instead (GH34522)

`DatetimeIndex.to_perioddelta()` is deprecated and will be removed in a future version. Use `index - index.to_period(freq).to_timestamp()` instead (GH34853)

`DataFrame.melt()` accepting a `value_name` that already exists is deprecated, and will be removed in a future version (GH34731)

The `center` keyword in the `DataFrame.expanding()` function is deprecated and will be removed in a future version (GH20647)

Performance improvements

Performance improvement in *Timedelta* constructor (GH30543)

Performance improvement in *Timestamp* constructor (GH30543)

Performance improvement in flex arithmetic ops between *DataFrame* and *Series* with `axis=0` (GH31296) • Performance improvement in arithmetic ops between *DataFrame* and *Series* with `axis=1` (GH33600)

The internal index method `_shallow_copy()` now copies cached attributes over to the new index, avoiding creating these again on the new index. This can speed up many operations that depend on creating copies of existing indexes (GH28584, GH32640, GH32669)

5.2. Version 1.1

Significant performance improvement when creating a *DataFrame* with

sparse values from `scipy.sparse` matrices using the `DataFrame.sparse.from_spmatrix()` constructor (GH32821, GH32825, GH32826, GH32856, GH32858).

Performance improvement for `groupby` methods `first()` and `last()` (GH34178)
Performance improvement in `factorize()` for nullable (integer and Boolean) dtypes (GH33064).

Performance improvement when constructing *Categorical* objects (GH33921) Fixed performance regression in `.qcut()` and `.cut()` (GH33921)

Performance improvement in reductions (sum, prod, min, max) for nullable (integer and Boolean) dtypes (GH30982, GH33261, GH33442).

Performance improvement in arithmetic operations between two *DataFrame* objects (GH32779)

Performance improvement in `.core.groupby.RollingGroupby` (GH34052)

Performance improvement in arithmetic operations (sub, add, mul, div) for *MultiIndex* (GH34297)

Performance improvement in `DataFrame[bool_indexer]` when `bool_indexer` is a list (GH33924)

Significant performance improvement of `io.formats.style.Styler.render()` with styles added with various ways such as `io.formats.style.Styler.apply()`, `io.formats.style.Styler.applymap()` or `io.formats.style.Styler.bar()` (GH19917)

Bug fixes

Categorical

Passing an invalid `fill_value` to `Categorical.take()` raises a `ValueError` instead of `TypeError` (GH33660)

Combining a *Categorical* with integer categories and which contains missing values with a float dtype column in operations such as `concat()` or `append()` will now result in a float column instead of an object dtype column (GH33607)

Bug where `merge()` was unable to join on non-unique categorical indices (GH28189) Bug when passing categorical data to *Index* constructor along with `dtype=object` incorrectly returning a *CategoricalIndex* instead of object-

dtype *Index* (GH32167) Bug where *Categorical* comparison operator `__ne__` would incorrectly evaluate to False when either element was missing (GH32276)

Categorical.fillna() now accepts *Categorical* other argument (GH32420)
Repr of *Categorical* was not distinguishing between int and str (GH33676)
Datetime-like

Passing an integer dtype other than int64 to `np.array(period_index, dtype=...)` will now raise `TypeError` instead of incorrectly using int64 (GH32255)
Series.to_timestamp() now raises a `TypeError` if the axis is not a *PeriodIndex*. Previously an `AttributeError` was raised (GH33327)
Series.to_period() now raises a `TypeError` if the axis is not a *DatetimeIndex*. Previously an `AttributeError` was raised (GH33327)
Period no longer accepts tuples for the freq argument (GH34658)
Bug in *Timestamp* where constructing a *Timestamp* from ambiguous epoch time and calling constructor again changed the *Timestamp.value()* property (GH24329)

DatetimeArray.searchsorted(), *TimedeltaArray.searchsorted()*, *PeriodArray.searchsorted()* not recognizing non- scalars and incorrectly raising `ValueError` instead of `TypeError` (GH30950)

Bug in *Timestamp* where constructing *Timestamp* with dateutil timezone less than 128 nanoseconds before daylight saving time switch from winter to summer would result in nonexistent time (GH31043)

Bug in *Period.to_timestamp()*, *Period.start_time()* with microsecond frequency returning a timestamp one nanosecond earlier than the correct time (GH31475)

Timestamp raised a confusing error message when year, month or day is missing (GH31200)

Bug in *DatetimeIndex* constructor incorrectly accepting bool-dtype inputs (GH32668)

Bug in *DatetimeIndex.searchsorted()* not accepting a list or *Series* as its argument (GH32762) • Bug where *PeriodIndex()* raised when passed a *Series* of strings (GH26109)

Bug in *Timestamp* arithmetic when adding or subtracting an `np.ndarray` with `timedelta64` dtype (GH33296)

Bug in *DatetimeIndex.to_period()* not inferring the frequency when called with no arguments (GH33358)

Bug in *DatetimeIndex.tz_localize()* incorrectly retaining `freq` in some cases where the original `freq` is no longer valid (GH30511)

Bug in *DatetimeIndex.intersection()* losing `freq` and `timezone` in some cases (GH33604) Bug in *DatetimeIndex.get_indexer()* where incorrect output would be returned for mixed datetimelike targets (GH33741)

Bug in *DatetimeIndex* addition and subtraction with some types of *DateOffset* objects incorrectly retaining an invalid `freq` attribute (GH33779)

Bug in *DatetimeIndex* where setting the `freq` attribute on an index could silently change the `freq` attribute on another index viewing the same data (GH33552)

DataFrame.min() and *DataFrame.max()* were not returning consistent results with *Series.min()* and *Series.max()* when called on objects initialized with empty `pd.to_datetime()`

Bug in *DatetimeIndex.intersection()* and *TimedeltaIndex.intersection()* with results not having the correct `name` attribute (GH33904)

Bug in *DatetimeArray.__setitem__()*, *TimedeltaArray.__setitem__()*, *PeriodArray.__setitem__()* incorrectly allowing values with `int64` dtype to be silently cast (GH33717)

Bug in subtracting *TimedeltaIndex* from *Period* incorrectly raising `TypeError` in some cases where it should succeed and `IncompatibleFrequency` in some cases where it should raise `TypeError` (GH33883)

Bug in constructing a *Series* or *Index* from a read-only NumPy array with non-ns resolution which converted to object dtype instead of coercing to `datetime64[ns]` dtype when within the timestamp bounds (GH34843).

The `freq` keyword in *Period*, *date_range()*, *period_range()*, `pd.tseries.frequencies.to_offset()` no longer allows tuples, pass as string instead (GH34703)

Bug in *DataFrame.append()* when appending a *Series* containing a scalar tz-aware *Timestamp* to an empty *DataFrame* resulted in an object column instead of `datetime64[ns, tz]` dtype (GH35038)

5.2. Version 1.1

OutOfBoundsDatetime issues an improved error message when timestamp is out of implementation bounds. (GH32967)

Bug in *AbstractHolidayCalendar.holidays()* when no rules were defined (GH31415) Bug in Tick comparisons raising *TypeError* when comparing against *timedelta*-like objects (GH34088)

Bug in Tick multiplication raising *TypeError* when multiplying by a float (GH34486) **Timedelta**

Bug in constructing a *Timedelta* with a high precision integer that would round the *Timedelta* components (GH31354)

Bug in dividing `np.nan` or `None` by *Timedelta* incorrectly returning `NaT` (GH31869) *Timedelta* now understands `µs` as an identifier for microsecond (GH32899)

Timedelta string representation now includes nanoseconds, when nanoseconds are non-zero (GH9309)

Bug in comparing a *Timedelta* object against an `np.ndarray` with `timedelta64` dtype incorrectly viewing all entries as unequal (GH33441)

Bug in *timedelta_range()* that produced an extra point on a edge case (GH30353, GH33498)

Bug in *DataFrame.resample()* that produced an extra point on a edge case (GH30353, GH13022, GH33498)

Bug in *DataFrame.resample()* that ignored the `loffset` argument when dealing with *timedelta* (GH7687, GH33498)

Bug in *Timedelta* and *.to_timedelta()* that ignored the `unit` argument for string input (GH12136)

Timezones

Bug in *to_datetime()* with `infer_datetime_format=True` where timezone

names (e.g. UTC) would not be parsed correctly (GH33133)

Numeric

Bug in *DataFrame.floordiv()* with `axis=0` not treating division-by-zero like *Series.floordiv()* (GH31271)

Bug in *to_numeric()* with string argument "uint64" and `errors="coerce"` silently fails (GH32394) • Bug in *to_numeric()* with `downcast="unsigned"` fails for empty data (GH32493)

Bug in *DataFrame.mean()* with `numeric_only=False` and either `datetime64` dtype or `PeriodDtype` column incorrectly raising `TypeError` (GH32426)

Bug in *DataFrame.count()* with `level="foo"` and index level "foo" containing NaNs causes segmentation fault (GH21824)

Bug in *DataFrame.diff()* with `axis=1` returning incorrect results with mixed dtypes (GH32995)

Bug in *DataFrame.corr()* and *DataFrame.cov()* raising when handling nullable integer columns with `.NA` (GH33803)

-

-

Bug in arithmetic operations between *DataFrame* objects with non-overlapping columns with duplicate labels causing an infinite loop (GH35194)

Bug in *DataFrame* and *Series* addition and subtraction between object-dtype objects and `datetime64` dtype objects (GH33824)

Bug in *Index.difference()* giving incorrect results when comparing a *Float64Index* and object *Index* (GH35217)

Bug in *DataFrame* reductions (e.g. `df.min()`, `df.max()`) with `ExtensionArray` dtypes (GH34520, GH32651)

Series.interpolate() and *DataFrame.interpolate()* now raise a `ValueError` if `limit_direction` is 'forward' or 'both' and `method` is 'backfill' or 'bfill' or `limit_direction` is 'backward' or 'both' and `method` is 'pad' or 'ffill' (GH34746)

Conversion

Bug in *Series* construction from NumPy array with big-endian datetime64 dtype (GH29684)

Bug in *Timedelta* construction with large nanoseconds keyword value

(GH32402) Bug in *DataFrame* construction where sets would be duplicated rather than raising (GH32582)

The *DataFrame* constructor no longer accepts a list of *DataFrame* objects. Because of changes to NumPy, *DataFrame* objects are now consistently treated as 2D objects, so a list of *DataFrame* objects is considered 3D, and no longer acceptable for the *DataFrame* constructor (GH32289).

Bug in *DataFrame* when initiating a frame with lists and assign columns with nested list for MultiIndex (GH32173)

Improved error message for invalid construction of list when creating a new index (GH35190)

Strings

Bug in the *astype()* method when converting “string” dtype data to nullable integer dtype (GH32450).

Fixed issue where taking min or max of a *StringArray* or *Series* with *StringDtype* type would raise. (GH31746)

Bug in *Series.str.cat()* returning NaN output when other had *Index* type (GH33425) *.api.dtypes.is_string_dtype()* no longer incorrectly identifies categorical series as string.

•

Interval

Bug in *IntervalArray* incorrectly allowing the underlying data to be changed when setting values

(GH32782)

5.2. Version 1.1

Indexing

DataFrame.xs() now raises a *TypeError* if a *level* keyword is supplied and the *axis* is not a *MultiIndex*. Previously an *AttributeError* was raised (GH33610)

Bug in slicing on a *DatetimeIndex* with a partial-timestamp dropping high-

resolution indices near the end of a year, quarter, or month (GH31064)
Bug in `PeriodIndex.get_loc()` treating higher-resolution strings differently from `PeriodIndex.get_value()` (GH31172)
Bug in `Series.at()` and `DataFrame.at()` not matching `.loc` behavior when looking up an integer in a `Float64Index` (GH31329)
Bug in `PeriodIndex.is_monotonic()` incorrectly returning `True` when containing leading `NaT` entries (GH31437)
Bug in `DatetimeIndex.get_loc()` raising `KeyError` with converted-integer key instead of the userpassed key (GH31425)
Bug in `Series.xs()` incorrectly returning `Timestamp` instead of `datetime64` in some object-dtype cases (GH31630)
Bug in `DataFrame.iat()` incorrectly returning `Timestamp` instead of `datetime` in some object-dtype cases (GH32809)
Bug in `DataFrame.at()` when either columns or index is non-unique (GH33041) Bug in `Series.loc()` and `DataFrame.loc()` when indexing with an integer key on a objectdtype `Index` that is not all-integers (GH31905)
Bug in `DataFrame.iloc.__setitem__()` on a `DataFrame` with duplicate columns incorrectly setting values for all matching columns (GH15686, GH22036)
Bug in `DataFrame.loc()` and `Series.loc()` with a `DatetimeIndex`, `TimedeltaIndex`, or `PeriodIndex` incorrectly allowing lookups of non-matching datetime-like dtypes (GH32650)
Bug in `Series.__getitem__()` indexing with non-standard scalars, e.g. `np.dtype` (GH32684)

-

Bug in `Index` constructor where an unhelpful error message was raised for NumPy scalars (GH33017)

Bug in `DataFrame.lookup()` incorrectly raising an `AttributeError` when `frame.index` or `frame.columns` is not unique; this will now raise a `ValueError` with a helpful error message (GH33041)

Bug in `Interval` where a `Timedelta` could not be added or subtracted from a `Timestamp` interval (GH32023)

Bug in `DataFrame.copy()` not invalidating `_item_cache` after copy caused post-copy value updates to not be reflected (GH31784)

Fixed regression in *DataFrame.loc()* and *Series.loc()* throwing an error when a *datetime64[ns, tz]* value is provided (GH32395)

Bug in *Series.__getitem__()* with an integer key and a *MultiIndex* with leading integer level failing to raise *KeyError* if the key is not present in the first level (GH33355) Bug in *DataFrame.iloc()* when slicing a single column *DataFrame* with *ExtensionDtype* (e.g. *df.iloc[:, :1]*) returning an invalid result (GH32957)

Bug in *DatetimeIndex.insert()* and *TimedeltaIndex.insert()* causing index freq to be lost when setting an element into an empty *Series* (GH33573)

Bug in *Series.__setitem__()* with an *IntervalIndex* and a list-like key of integers (GH33473) *Series.__getitem__()* allowing missing labels with *np.ndarray*, *Index*, *Series* indexers but not list, these now all raise *KeyError* (GH33646)

Bug in *DataFrame.truncate()* and *Series.truncate()* where index was assumed to be monotone increasing (GH33756)

Indexing with a list of strings representing datetimes failed on *DatetimeIndex* or *PeriodIndex* (GH11278)

Bug in *Series.at()* when used with a *MultiIndex* would raise an exception on valid inputs (GH26989)

Bug in *DataFrame.loc()* with dictionary of values changes columns with dtype of int to float (GH34573)

Bug in *Series.loc()* when used with a *MultiIndex* would raise an *IndexingError* when accessing a *None* value (GH34318)

Bug in *DataFrame.reset_index()* and *Series.reset_index()* would not preserve data types on an empty *DataFrame* or *Series* with a *MultiIndex* (GH19602)

•

Bug in *Series* and *DataFrame* indexing with a time key on a *DatetimeIndex* with *NaT* entries

(GH35114)

Missing

Calling *fillna()* on an empty *Series* now correctly returns a shallow copied object. The behaviour is now consistent with *Index*, *DataFrame* and a non-empty *Series* (GH32543).

Bug in *Series.replace()* when argument *to_replace* is of type dict/list and is used on a *Series* containing <NA> was raising a *TypeError*. The method now handles this by ignoring <NA> values when doing the comparison for the replacement (GH32621)

Bug in *any()* and *all()* incorrectly returning <NA> for all False or all True values using the nullable Boolean dtype and with *skipna=False* (GH33253)
Clarified documentation on *interpolate* with *method=akima*. The *der* parameter must be scalar or None (GH33426)

DataFrame.interpolate() uses the correct axis convention now. Previously interpolating along columns lead to interpolation along indices and vice versa. Furthermore interpolating with methods *pad*, *ffill*, *bfill* and *backfill* are identical to using these methods with *DataFrame.fillna()* (GH12918, GH29146)

Bug in *DataFrame.interpolate()* when called on a *DataFrame* with column names of string type was throwing a *ValueError*. The method is now independent of the type of the column names (GH33956)

Passing NA into a format string using format specs will now work. For example "{:.1f}".format(pd. NA) would previously raise a *ValueError*, but will now return the string "<NA>" (GH34740)

Bug in *Series.map()* not raising on invalid *na_action* (GH32815)

5.2. Version 1.1

MultiIndex

DataFrame.swaplevels() now raises a *TypeError* if the axis is not a *MultiIndex*. Previously an *AttributeError* was raised (GH31126)

Bug in *Dataframe.loc()* when used with a *MultiIndex*. The returned values were not in the same order as the given inputs (GH22797)

•

In [80]: df = pd.DataFrame(np.arange(4),

....: index=[["a", "a", "b", "b"], [1, 2, 1, 2]])

....:

Rows are now ordered as the requested keys

In [81]: df.loc([('b', 'a'), [2, 1]], :) **Out[81]:**

b 2 3

2a 2 1

1 0

[4 rows x 1 columns]

Bug in `MultiIndex.intersection()` was not guaranteed to preserve order when `sort=False`. (GH31325)

Bug in `DataFrame.truncate()` was dropping *MultiIndex* names. (GH34564)

In [82]: left = pd.MultiIndex.from_arrays([["b", "a"], [2, 1]])

In [83]: right = pd.MultiIndex.from_arrays([["a", "b", "c"], [1, 2, 3]])

Common elements are now guaranteed to be ordered by the left side **In**

[84]: left.intersection(right, sort=False) **Out[84]:**

```
MultiIndex([('b', 2),  
            ('a', 1)],  
            )
```

Bug when joining two *MultiIndex* without specifying level with different columns. `Returnindexers` parameter was ignored. (GH34074)

IO

Passing a set as `names` argument to `.read_csv()`, `.read_table()`, or `.read_fwf()` will raise `ValueError: Names should be an ordered collection`. (GH34946)

Bug in print-out when `display.precision` is zero. (GH20359)

Bug in `read_json()` where integer overflow was occurring when json contains big number strings. (GH30320)

`read_csv()` will now raise a `ValueError` when the arguments `header` and `prefix` both are not `None`. (GH27394)

•

Bug in `DataFrame.to_json()` was raising `NotFoundError` when `path_or_buf` was an S3 URI (GH28375)

`DataFrame.to_parquet()` overwriting pyarrow's default for `coerce_timestamps`; following pyarrow's default allows writing nanosecond

timestamps with version="2.0" (GH31652). Bug in `read_csv()` was raising `TypeError` when `sep=None` was used in combination with `comment` keyword (GH31396)

Bug in `HDFStore` that caused it to set to `int64` the `dtype` of a `datetime64` column when reading a `DataFrame` in Python 3 from fixed format written in Python 2 (GH31750) `read_sas()` now handles dates and datetimes larger than `Timestamp.max` returning them as `datetime.datetime` objects (GH20927)

Bug in `DataFrame.to_json()` where `Timedelta` objects would not be serialized correctly with `date_format="iso"` (GH28256)

`read_csv()` will raise a `ValueError` when the column names passed in `parse_dates` are missing in the `Dataframe` (GH31251)

Bug in `read_excel()` where a UTF-8 string with a high surrogate would cause a segmentation violation (GH23809)

Bug in `read_csv()` was causing a file descriptor leak on an empty file (GH31488) Bug in `read_csv()` was causing a segfault when there were blank lines between the header and data rows (GH28071)

Bug in `read_csv()` was raising a misleading exception on a permissions issue (GH23784)

Bug in `read_csv()` was raising an `IndexError` when `header=None` and two extra data columns

Bug in `read_sas()` was raising an `AttributeError` when reading files from Google Cloud Storage (GH33069)

Bug in `DataFrame.to_sql()` where an `AttributeError` was raised when saving an out of bounds date (GH26761)

Bug in `read_excel()` did not correctly handle multiple embedded spaces in OpenDocument text cells. (GH32207)

Bug in `read_json()` was raising `TypeError` when reading a list of `Booleans` into a `Series`. (GH31464)

Bug in `.io.json.json_normalize()` where location specified by `record_path` doesn't point to an array. (GH26284)

-

`.read_hdf()` has a more explicit error message when loading an unsupported HDF file (GH9539)

Bug in `read_feather()` was raising an `ArrowIOError` when reading an s3 or http file path (GH29055)

Bug in *to_excel()* could not handle the column name render and was raising an `KeyError` (GH34331)

Bug in *execute()* was raising a `ProgrammingError` for some DB-API drivers when the SQL statement contained the `%` character and no parameters were present (GH34211) Bug in *StataReader()* which resulted in categorical variables with different dtypes when reading data using an iterator. (GH31544)

HDFStore.keys() has now an optional `include` parameter that allows the retrieval of all native HDF5 table names (GH29916)

`TypeError` exceptions raised by *read_csv()* and *read_table()* were showing as `parser_f` when an unexpected keyword argument was passed (GH25648)

Bug in *read_excel()* for ODS files removes 0.0 values (GH27222)

5.2. Version 1.1

ujson.encode() was raising an `OverflowError` with numbers larger than `sys.maxsize` (GH34395)

Bug in *HDFStore.append_to_multiple()* was raising a `ValueError` when the `min_itemsize` parameter is set (GH11238)

Bug in *create_table()* now raises an error when `column` argument was not specified in `data_columns` on input (GH28156)

read_json() now could read line-delimited json file from a file url while `lines` and `chunksize` are set.

Bug in *DataFrame.to_sql()* when reading DataFrames with `-np.inf` entries with MySQL now has a more explicit `ValueError` (GH34431)

Bug where capitalised files extensions were not decompressed by *read_** functions (GH35164)

Bug in *read_excel()* that was raising a `TypeError` when `header=None` and `index_col` is given as a list (GH31783)

Bug in *read_excel()* where datetime values are used in the header in a *MultiIndex* (GH34748)

read_excel() no longer takes `**kwds` arguments. This means that passing in the keyword argument `chunksize` now raises a `TypeError` (previously raised a `NotImplementedError`), while passing in the keyword argument `encoding` now raises a `TypeError` (GH34464)

Bug in *DataFrame.to_records()* was incorrectly losing timezone information in `timezoneaware` `datetime64` columns (GH32535)

Plotting

DataFrame.plot() for line/bar now accepts color by dictionary (GH8193).

Bug in *DataFrame.plot.hist()* where weights are not working for multiple columns (GH33173)

Bug in *DataFrame.boxplot()* and *DataFrame.plot.boxplot()* lost color attributes of medianprops, whiskerprops, capprops and boxprops (GH30346)

Bug in *DataFrame.hist()* where the order of column argument was ignored (GH29235) Bug in *DataFrame.plot.scatter()* that when adding multiple plots with different cmap, colorbars always use the first cmap (GH33389)

Bug in *DataFrame.plot.scatter()* was adding a colorbar to the plot even if the argument c was assigned to a column containing color names (GH34316)

Bug in *.plotting.bootstrap_plot()* was causing cluttered axes and overlapping labels (GH34905)

569 Chapter 5. Release notes

Bug in *DataFrame.plot.scatter()* caused an error when plotting variable marker sizes (GH32904)

GroupBy/resample/rolling

Using a *.api.indexers.BaseIndexer* with count, min, max, median, skew, cov, corr will now return correct results for any monotonic

.api.indexers.BaseIndexer descendant (GH32865)

DataFrameGroupby.mean() and *SeriesGroupby.mean()* (and similarly for *median()*, *std()* and *var()*) now raise a *TypeError* if a non-accepted keyword argument is passed into it. Previously an

UnsupportedFunctionCall was raised (*AssertionError* if *min_count* passed into *median()*) (GH31485)

GroupBy.apply() raises *ValueError* when the by axis is not sorted, has duplicates, and the applied func does not mutate passed in objects (GH30667)

Bug in *DataFrameGroupBy.transform()* produces an incorrect result with transformation functions (GH30918)

Bug in *Groupby.transform()* was returning the wrong result when grouping by multiple keys of which some were categorical and others not (GH32494)

Bug in *GroupBy.count()* causes segmentation fault when grouped-by

columns contain NaNs (GH32841)

Bug in *DataFrame.groupby()* and *Series.groupby()* produces inconsistent type when aggregating Boolean Series (GH32894)

Bug in *DataFrameGroupBy.sum()* and *SeriesGroupBy.sum()* where a large negative number would be returned when the number of non-null values was below *min_count* for nullable integer dtypes (GH32861) • Bug in *SeriesGroupBy.quantile()* was raising on nullable integers (GH33136)

Bug in *DataFrame.resample()* where an *AmbiguousTimeError* would be raised when the resulting timezone aware *DatetimeIndex* had a DST transition at midnight (GH25758) Bug in *DataFrame.groupby()* where a *ValueError* would be raised when grouping by a categorical column with read-only categories and *sort=False* (GH33410) Bug in *GroupBy.agg()*, *GroupBy.transform()*, and *GroupBy.resample()* where subclasses are not preserved (GH28330)

Bug in *SeriesGroupBy.agg()* where any column name was accepted in the named aggregation of *SeriesGroupBy* previously. The behaviour now allows only str and callables else would raise *TypeError*. (GH34422)

Bug in *DataFrame.groupby()* lost the name of the *Index* when one of the agg keys referenced an empty list (GH32580)

Bug in *Rolling.apply()* where *center=True* was ignored when *engine='numba'* was specified (GH34784)

Bug in *DataFrame.ewm.cov()* was throwing *AssertionError* for *MultiIndex* inputs (GH34440)

Bug in *core.groupby.DataFrameGroupBy.quantile()* raised *TypeError* for non-numeric types rather than dropping the columns (GH27892)

Bug in *core.groupby.DataFrameGroupBy.transform()* when *func='nunique'* and columns are of type *datetime64*, the result would also be of type *datetime64* instead of *int64* (GH35109)

Bug in *DataFrame.groupby()* raising an *AttributeError* when selecting a column and aggregating with *as_index=False* (GH35246).

Bug in *DataFrameGroupBy.first()* and *DataFrameGroupBy.last()* that would raise an unnecessary *ValueError* when grouping on multiple Categoricals

(GH34951)

5.2. Version 1.1

Reshaping

Bug effecting all numeric and Boolean reduction methods not returning subclassed data type. (GH25596)

Bug in *DataFrame.pivot_table()* when only MultiIndexed columns is set (GH17038) Bug in *DataFrame.unstack()* and *Series.unstack()* can take tuple names in MultiIndexed data (GH19966)

Bug in *DataFrame.pivot_table()* when margin is True and only column is defined (GH31016)

Fixed incorrect error message in *DataFrame.pivot()* when columns is set to None. (GH30924)

Bug in *crosstab()* when inputs are two *Series* and have tuple names, the output will keep a dummy *MultiIndex* as columns. (GH18321)

DataFrame.pivot() can now take lists for index and columns arguments (GH21425) Bug in *concat()* where the resulting indices are not copied when copy=True (GH29879)

Bug in *SeriesGroupBy.aggregate()* was resulting in aggregations being overwritten when they shared the same name (GH30880)

Bug where *Index.astype()* would lose the name attribute when converting from *Float64Index* to *Int64Index*, or when casting to an *ExtensionArray* dtype (GH32013) *Series.append()* will now raise a *TypeError* when passed a *DataFrame* or a sequence containing *DataFrame* (GH31413)

DataFrame.replace() and *Series.replace()* will raise a *TypeError* if *to_replace* is not an expected type. Previously the replace would fail silently (GH18634)

Bug on inplace operation of a *Series* that was adding a column to the *DataFrame* from where it was originally dropped from (using *inplace=True*) (GH30484)

Bug in *DataFrame.apply()* where callback was called with *Series* parameter even though *raw=True* requested. (GH32423)

Bug in *DataFrame.pivot_table()* losing timezone information when creating a *MultiIndex* level from a column with timezone-aware dtype (GH32558)

Bug in *concat()* where when passing a non-dict mapping as objs would raise a `TypeError` (GH32863)

DataFrame.agg() now provides more descriptive `SpecificationError` message when attempting to aggregate a non-existent column (GH32755)

Bug in *DataFrame.unstack()* when *MultiIndex* columns and *MultiIndex* rows were used (GH32624, GH24729 and GH28306)

Appending a dictionary to a *DataFrame* without passing `ignore_index=True` will raise `TypeError: Can only append a dict if ignore_index=True` instead of `TypeError: Can only append a :class:`Series` if ignore_index=True or if the :class:`Series` has a name` (GH30871)

Bug in *DataFrame.corrwith()*, *DataFrame.memory_usage()*, *DataFrame.dot()*, *DataFrame.idxmin()*, *DataFrame.idxmax()*, *DataFrame.duplicated()*, *DataFrame.isin()*, *DataFrame.count()*, *Series.explode()*, *Series.asof()* and *DataFrame.asof()* not returning subclassed types. (GH31331)

Bug in *concat()* was not allowing for concatenation of *DataFrame* and *Series* with duplicate keys (GH33654)

Bug in *cut()* raised an error when the argument labels contains duplicates (GH33141) Ensure only named functions can be used in *eval()* (GH32460)

Dataframe.aggregate() and *Series.aggregate()* was causing a recursive loop in some cases (GH34224)

Fixed bug in *melt()* where melting *MultiIndex* columns with `col_level > 0` would raise a `KeyError` on `id_vars` (GH34129)

Bug in *Series.where()* with an empty *Series* and empty cond having non-bool dtype (GH34592)

Fixed regression where *DataFrame.apply()* would raise `ValueError` for elements with `S` dtype

(GH34529)

Sparse

Creating a *SparseArray* from timezone-aware dtype will issue a warning

before dropping timezone information, instead of doing so silently (GH32501)

Bug in `arrays.SparseArray.from_spmatrix()` wrongly read scipy sparse matrix (GH31991)

Bug in `Series.sum()` with `SparseArray` raised a `TypeError` (GH25777)

Bug where `DataFrame` containing an all-sparse `SparseArray` filled with NaN when indexed by a list-like (GH27781, GH29563)

The repr of `SparseDtype` now includes the repr of its `fill_value` attribute.

Previously it used `fill_value`'s string representation (GH34352)

Bug where empty `DataFrame` could not be cast to `SparseDtype` (GH33113)

Bug in `arrays.SparseArray()` was returning the incorrect type when indexing a sparse dataframe with an iterable (GH34526, GH34540)

ExtensionArray

Fixed bug where `Series.value_counts()` would raise on empty input of `Int64` dtype (GH33317)

Fixed bug in `concat()` when concatenating `DataFrame` objects with non-overlapping columns resulting in object-dtype columns rather than preserving the extension dtype (GH27692, GH33027)

Fixed bug where `StringArray.isna()` would return `False` for NA values when `.options.mode.use_inf_as_na` was set to `True` (GH33655)

Fixed bug in `Series` construction with `EA` dtype and index but no data or scalar data fails (GH26469)

Fixed bug that caused `Series.__repr__()` to crash for extension types whose elements are multidimensional arrays (GH33770).

Fixed bug where `Series.update()` would raise a `ValueError` for `ExtensionArray` dtypes with missing values (GH33980)

Fixed bug where `StringArray.memory_usage()` was not implemented (GH33963)

Fixed bug where `DataFrameGroupBy()` would ignore the `min_count` argument for aggregations on nullable Boolean dtypes (GH34051)

Fixed bug where the constructor of `DataFrame` with `dtype='string'` would fail (GH27953, GH33623)

Bug where `DataFrame` column set to scalar extension type was considered an object type rather than the extension type (GH34832)

Fixed bug in `IntegerArray.astype()` to correctly copy the mask as well (GH34931). **5.2. Version 1.1**

Other

Set operations on an object-dtype *Index* now always return object-dtype results (GH31401)

Fixed `.testing.assert_series_equal()` to correctly raise if the left argument is a different subclass with `check_series_type=True` (GH32670).

Getting a missing attribute in a `DataFrame.query()` or `DataFrame.eval()` string raises the correct `AttributeError` (GH32408)

Fixed bug in `.testing.assert_series_equal()` where dtypes were checked for `Interval` and `ExtensionArray` operands when `check_dtype` was `False` (GH32747)

Bug in `DataFrame.__dir__()` caused a segfault when using unicode surrogates in a column name (GH25509)

Bug in `DataFrame.equals()` and `Series.equals()` in allowing subclasses to be equal (GH34402).

Contributors

A total of 368 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

3vts +

A Brooks +

Abbie Popa +

Achmad Syarif Hidayatullah +

Adam W Bagaskarta +

Adrian Mastronardi +

Aidan Montare +

Akbar Septriyan +

Akos Furton +

Alejandro Hall +

Alex Hall +

-
-
-
-

-
-

Alex Itkes +
Alex Kirko
Ali McMaster +
Alvaro Aleman +
Amy Graham +
Andrew Schonfeld +
Andrew Shumanskiy +
Andrew Wieteska + Angela Ambroz Anjali Singh +
Anna Daglis
Anthony Milbourne +
Antony Lee +
Ari Sosnovsky +
Arkadeep Adhikari +
Arunim Samudra +
Ashkan +
Ashwin Prakash Nalwade +
Ashwin Srinath +
Atsushi Nukariya +
Ayappan +
Ayla Khan +
Bart +
Bart Broere +

-
-

•
Benjamin Beier Liu +

Benjamin Fischer +
Bharat Raghunathan Bradley Dice +
Brendan Sullivan +
Brian Strand +
Carsten van Weelden + Chamoun Saoma +
ChrisRobo +

Christian Chwala
Christopher Whelan
Christos Petropoulos + Chuanzhu Xu
CloseChoice +
Clément Robert +
CuylenE +
DanBasson +
Daniel Saxton
Danilo Horta +
DavaIlhamHaeruzaman + Dave Hirschfeld
Dave Hughes
David Rouquet +

-
-
-

David S +

Deepyaman Datta
Dennis Bakhuis +
Derek McCammond +
Devjeet Roy +
Diane Trout
Dina +
Dom +
Drew Seibert +
EdAbati
Emiliano Jordan +
Erfan Nariman +
Eric Groszman +
Erik Hasse +
Erkam Uyanik +
Evan D +
Evan Kanter +
Fangchen Li +
Farhan Reynaldo +

Farhan Reynaldo Hutabarat + Florian Jetter +
Fred Reiss +
GYHHAHA +

-
-
-

Gabriel Moreira +

Gabriel Tutui +
Galuh Sahid
Gaurav Chauhan +
George Hartzell +
Gim Seng +
Giovanni Lanzani +
Gordon Chen +
Graham Wetzler + Guillaume Lemaitre Guillem Sánchez +
HH-MWB +
Harshavardhan Bachina
How Si Wei
Ian Eaves
Iqrar Agalosi Nureyza +
Irv Lustig
Iva Laginja +
JDkuba
Jack Greisman +
Jacob Austin +
Jacob Deppen +
Jacob Peacock +
Jake Tae +

-
-
-

Jake Vanderplas +

James Cobon-Kerr
Jan Cervenka +
Jan Škoda
Jane Chen +
Jean-Francois Zinque +
Jeanderson Barros Candido + Jeff Reback
Jered Dominguez-Trujillo + Jeremy Schendel
Jesse Farnham • Jiaxiang Jihwan Song +
Joaquim L. Viegas +
Joel Nothman
John Bodley +
John Paton +
Jon Thielen +
Joris Van den Bossche
Jose Manuel Martí +
Joseph Gulian +
Josh Dimarsky
Joy Bhalla +
João Veiga +

•

•

•

Julian de Ruiter +

Justin Essert +
Justin Zheng
KD-dev-lab +
Kaiqi Dong
Karthik Mathur + Kaushal Rohit +
Kee Chong Tan
Ken Mankoff +
Kendall Masse
Kenny Huynh +
Ketan +
Kevin Anderson + Kevin Bowey +

Kevin Sheppard Kilian Lieret +
Koki Nishihara + Krishna Chivukula + KrishnaSai2020 + Lesley +
Lewis Cowles +
Linda Chen +
Linxiao Wu +

-
-

•
Lucca Delchiaro Costabile +

MBrouns +
Mabel Villalba
Mabroor Ahmed +
Madhuri Palanivelu +
Mak Sze Chun
Malcolm +
Marc Garcia
Marco Gorelli
Marian Denes +
Martin Bjeldbak Madsen + Martin Durant +
Martin Fleischmann +
Martin Jones +
Martin Winkel
Martina Oefelein +
Marvzinc +
María Marino +
Matheus Cardoso +
Mathis Felardos +
Matt Roeschke
Matteo Felici +
Matteo Santamaria +

-
-
-

Matthew Roeschke

Matthias Bussonnier

Max Chen

Max Halford +

Mayank Bisht +

Megan Thong +

Michael Marino +

Miguel Marques +

Mike Kutzma

Mohammad Hasnain Mohsin Rajan + Mohammad Jafar Mashhadi +

MomIsBestFriend

Monica +

Natalie Jann

Nate Armstrong +

Nathanael +

Nick Newman +

Nico Schlömer +

Niklas Weber +

ObliviousParadigm +

Olga Lyashevskaya +

OlivierLuG +

Development Team

•

•

•

Parallels +

Patrick +

Patrick Cando +

Paul Lilley +

Paul Sanders +

PearceKieser +

Pedro Larroy +

Pedro Reis

Peter Bull +
Peter Steinbach +
Phan Duc Nhat Minh + Phil Kirlin +
Pierre-Yves Bourguignon + Piotr Kasprzyk +
Piotr Niełacny +
Prakhar Pandey
Prashant Anand +
Puneetha Pai +
Quang Nguyn +
Rafael Jaimes III +
Rafif +
RaisaDZ +
Rakshit Naidu +

-
-
-

Ram Rachum +

Red +
Ricardo Alanis +
Richard Shadrach +
Rik-de-Kort
Robert de Vries
Robin to Roxel +
Roger Erens +
Rohith295 +
Roman Yurchak • Ror +
Rushabh Vasani
Ryan
Ryan Nazareth
SAI SRAVAN MEDICHERLA + SHUBH CHATTERJEE + Sam Cohan
Samira-g-js +
Sandu Ursu +
Sang Agung +
SanthoshBala18 +

Sasidhar Kasturi +
SatheeshKumar Mohan +
Saul Shanabrook • Scott Gigante +
Sebastian Berg +

-
-
-

Sebastián Vanrell

Sergei Chipiga + Sergey +
ShilpaSugan +
Simon Gibbons Simon Hawkins Simon Legner + Soham Tiwari + Song
Wenhao + Souvik Mandal
Spencer Clark
Steffen Rehberg + Steffen Schmitz + Stijn Van Hoey
Stéphan Taljaard SultanOrazbayev + Sumanau Sareen SurajH1 +
Suvayu Ali +
Terji Petersen
Thomas J Fan + Thomas Li
Thomas Smith +

-
-
-

Tim Swast

Tobias Pitters +
Tom +
Tom Augspurger
Uwe L. Korn
Valentin Iovene +
Vandana Iyer +
Venkatesh Datta + Vijay Sai Mutyala + Vikas Pandey
Vipul Rai +
Vishwam Pandya + Vladimir Berkutov + Will Ayd

Will Holmgren
William +
William Ayd
Yago González +
Yosuke KOBAYASHI + Zachary Lawrence + Zaky Bilfagih +
Zeb Nicholls +
alimcmaster1

•

•

•

alm +

andhikayusup +
andresmcneill +
avinashpancham +
benabel +
bernie gray +
biddwan09 +
brock +
chris-b1
cleconte987 + • dan1261 +
david-cortes +
davidwales + dequadrass + dhuettenmoser + dilex42 + elmonsomiat +
epizzigoni + fjetter
gabrielvf1 + gdex1 +
gfyounG
guru kiran +
h-vishal
iamshwin
jamin-aws-ospo +
jbrockmendel
jfcorbett +
jnecus +
kernc

•

-

-

kota matsuoka +

kylekeppler +

leandermaben + • link2xt + manoj_koneni +

marydmit +

masterpiga +

maxime.song + • mglasder + moaraccounts +

mproszewska

neilkg

nrebena

ossdev07 +

paihu

pan Jacek +

partev +

patrick +

pedrooa +

pizzathief + proost

pvanhauw +

rbenes

-

-

-

rebecca-palmer rhshadrach + rjfs + s-scherrer +

sage +

sagungrp +

saalem3358 +

saloni30 +

smartswdeveloper +

smartvinnetou +

themien +

timhunderwood +

tolhassianipar +

tonywu1999
tsvikas
tv3141
venkateshdatta1993 +
vivikelapoutre +
willbowditch +
willpeppo +
za +
zaki-indra +

5.3 Version 1.0

5.3.1 What's new in 1.0.5 (June 17, 2020)

These are the changes in 1.0.5. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fix regression in *read_parquet()* when reading from file-like objects (GH34467). Fix regression in reading from public S3 buckets (GH34626).

Note this disables the ability to read Parquet files from directories on S3 again (GH26388, GH34632), which was added in the 1.0.4 release, but is now targeted for 1.1.0.

Fixed regression in *replace()* raising an `AssertionError` when replacing values in an extension dtype with values of a different dtype (GH34530)

Bug fixes

Fixed building from source with Python 3.8 fetching the wrong version of NumPy (GH34666)

Contributors

A total of 8 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Joris Van den Bossche
MeeseeksMachine
Natalie Jann +

Development Team
Simon Hawkins

Tom Augspurger

William Ayd

alimcmaster1

5.3.2 What's new in 1.0.4 (May 28, 2020)

These are the changes in 1.0.4. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fix regression where *Series.isna()* and *DataFrame.isna()* would raise for categorical dtype when *.options.mode.use_inf_as_na* was set to True (GH33594)

Fix regression in *GroupBy.first()* and *GroupBy.last()* where None is not preserved in object dtype (GH32800)

Fix regression in *DataFrame* reductions using *numeric_only=True* and *ExtensionArrays* (GH33256).

Fix performance regression in *memory_usage(deep=True)* for object dtype (GH33012)

Fix regression where *Categorical.replace()* would replace with NaN whenever the new value and replacement value were equal (GH33288)

Fix regression where an ordered *Categorical* containing only NaN values would raise rather than returning NaN when taking the minimum or maximum (GH33450)

Fix regression in *DataFrameGroupBy.agg()* with dictionary input losing *ExtensionArray* dtypes (GH32194)

Fix to preserve the ability to index with the “nearest” method with *xarray*'s *CftimeIndex*, an *Index* subclass (pydata/xarray#3751, GH32905).

Fix regression in *DataFrame.describe()* raising *TypeError: unhashable type: 'dict'* (GH32409)

Fix regression in *DataFrame.replace()* casts columns to object dtype if items in *to_replace* not in values (GH32988)

Fix regression in *Series.groupby()* would raise *ValueError* when grouping by *PeriodIndex* level

(GH34010)

5.3. Version 1.0

Fix regression in `GroupBy.rolling.apply()` ignores args and kwargs parameters (GH33433)

Fix regression in error message with `np.min` or `np.max` on unordered *Categorical* (GH33115)

Fix regression in `DataFrame.loc()` and `Series.loc()` throwing an error when a `datetime64[ns, tz]` value is provided (GH32395)

Bug fixes

Bug in `SeriesGroupBy.first()`, `SeriesGroupBy.last()`, `SeriesGroupBy.min()`, and `SeriesGroupBy.max()` returning floats when applied to nullable Booleans (GH33071) Bug in `Rolling.min()` and `Rolling.max()`: Growing memory usage after multiple calls when using a fixed window (GH30726)

Bug in `to_parquet()` was not raising `PermissionError` when writing to a private s3 bucket with invalid creds. (GH27679)

Bug in `to_csv()` was silently failing when writing to an invalid s3 bucket. (GH32486) Bug in `read_parquet()` was raising a `FileNotFoundError` when passed an s3 directory path. (GH26388)

Bug in `to_parquet()` was throwing an `AttributeError` when writing a partitioned parquet file to s3 (GH27596)

Bug in `GroupBy.quantile()` causes the quantiles to be shifted when the by axis contains NaN (GH33200,

GH33569)

Contributors

A total of 18 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Daniel Saxton

JDkuba +

Joris Van den Bossche

Kaiqi Dong

Mabel Villalba

MeeseeksMachine

MomIsBestFriend

Development Team

Simon Hawkins

Spencer Clark +

Tom Augspurger
Vikas Pandey +
alimcmaster1
h-vishal +
jbrockmendel
mproszewska +

•

neilkg +
rebecca-palmer +
5.3.3 What's new in 1.0.3 (March 17, 2020)

These are the changes in 1.0.3. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in `resample.agg` when the underlying data is non-writeable (GH31710) Fixed regression in *DataFrame* exponentiation with reindexing (GH32685) **Bug fixes**

Contributors

A total of 5 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

MeeseeksMachine
Development Team
Tom Augspurger
William Ayd
jbrockmendel

5.3.4 What's new in 1.0.2 (March 12, 2020)

These are the changes in 1.0.2. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Groupby

Fixed regression in *groupby(..).agg()* which was failing on frames with *MultiIndex* columns and a custom function (GH31777)
Fixed regression in *groupby(..).rolling(..).apply()* (*RollingGroupby*) where the raw parameter was ignored (GH31754)
Fixed regression in *rolling(..).corr()* when using a time offset (GH31789)
Fixed regression in *groupby(..).nunique()* which was modifying the original values if NaN values were present (GH31950)

Fixed regression in *DataFrame.groupby* raising a *ValueError* from an internal operation (GH31802)

Fixed regression in *groupby(..).agg()* calling a user-provided function an extra time on an empty input

(GH31760)

I/O

5.3. Version 1.0

Fixed regression in *read_csv()* in which the encoding option was not recognized with certain file-like objects (GH31819)

Fixed regression in *DataFrame.to_excel()* when the columns keyword argument is passed (GH31677)

Fixed regression in *ExcelFile* where the stream passed into the function was closed by the destructor. (GH31467)

Fixed regression where *read_pickle()* raised a *UnicodeDecodeError* when reading a py27 pickle with *MultiIndex* column (GH31988).

Reindexing/alignment

Fixed regression in *Series.align()* when other is a *DataFrame* and method is not None (GH31785)

Fixed regression in *DataFrame.reindex()* and *Series.reindex()* when reindexing with (tzaware) index and method=nearest (GH26683)

Fixed regression in *DataFrame.reindex_like()* on a *DataFrame* subclass raised an *AssertionError* (GH31925)

Fixed regression in *DataFrame* arithmetic operations with mis-matched columns (GH31623)

Other

Fixed regression in joining on *DatetimeIndex* or *TimedeltaIndex* to preserve freq in simple cases (GH32166)

Fixed regression in *Series.shift()* with datetime64 dtype when passing an integer fill_value (GH32591)

Fixed regression in the repr of an object-dtype *Index* with bools and missing values (GH32146)

Indexing with nullable boolean arrays

Previously indexing with a nullable Boolean array containing NA would raise a *ValueError*, however this is now permitted with NA being treated as *False*. (GH31503)

```
In [1]: s = pd.Series([1, 2, 3, 4])
```

```
In [2]: mask = pd.array([True, True, False, None], dtype="boolean") In [3]: s
```

```
Out[3]:
```

```
1  
2  
3  
4
```

```
Length: 4, dtype: int64
```

```
In [4]: mask Out[4]:
```

```
<BooleanArray>
```

```
[True, True, False, <NA>]
```

```
Length: 4, dtype: boolean
```

1.0.0-1.0.1

```
>>> s[mask]
```

```
Traceback (most recent call last):
```

```
(continues on next page)
```

```
(continued from previous page)
```

```
... ValueError: cannot mask with array containing NA / NaN values
```

1.0.2

```
In [5]: s[mask] Out[5]:
```

```
1  
2
```

```
Length: 2, dtype: int64
```

Bug fixes

Datetimelike

Bug in *Series.astype()* not copying for tz-naive and tz-aware datetime64 dtype (GH32490)

Bug where *to_datetime()* would raise when passed *pd.NA* (GH32213)
Improved error message when subtracting two *Timestamp* that result in an out-ofbounds *Timedelta*

(GH31774)
Categorical

Fixed bug where *Categorical.from_codes()* improperly raised a *ValueError* when passed nullable integer codes. (GH31779)

Fixed bug where *Categorical()* constructor would raise a *TypeError* when given a numpy array containing *pd.NA*. (GH31927)

Bug in *Categorical* that would ignore or crash when calling *Series.replace()* with a listlike *to_replace* (GH31720)

I/O

Using *pd.NA* with *DataFrame.to_json()* now correctly outputs a null value instead of an empty object (GH31615)

Bug in *.json_normalize()* when value in meta path is not iterable (GH31507)

Fixed pickling of *.NA*. Previously a new object was returned, which broke computations relying on *NA* being a singleton (GH31847)

Fixed bug in parquet roundtrip with nullable unsigned integer dtypes (GH31896). Experimental dtypes

Fixed bug in *DataFrame.convert_dtypes()* for columns that were already using the "string" dtype (GH31731).

Fixed bug in *DataFrame.convert_dtypes()* for series with mix of integers and strings (GH32117)

Fixed bug in *DataFrame.convert_dtypes()* where *BooleanDtype* columns were converted to *Int64* (GH32287)

Fixed bug in setting values using a slice indexer with string dtype (GH31772)

Fixed bug where *.core.groupby.GroupBy.first()* and *.core.groupby.*

GroupBy.last() would raise a *TypeError* when groups contained *pd.NA* in a column of object dtype (GH32123)

5.3. Version 1.0

Fixed bug where *DataFrameGroupBy.mean()*,

DataFrameGroupBy.median(),
DataFrameGroupBy.var(), and DataFrameGroupBy.std() would raise a
TypeError on Int64 dtype columns (GH32219)

Strings

Using pd.NA with *Series.str.repeat()* now correctly outputs a null value
instead of raising error for vector inputs (GH31632)

Rolling
Fixed rolling operations with variable window (defined by time duration) on
decreasing time index (GH32385).

Contributors

A total of 25 people contributed patches to this release. People with a “+” by
their names contributed a patch for the first time.

Anna Daglis +
Daniel Saxton
Irv Lustig
Jan Škoda
Joris Van den Bossche
Justin Zheng
Kaiqi Dong
Kendall Masse
Marco Gorelli
Matthew Roeschke
MeeseeksMachine
MomIsBestFriend

Development Team
Pedro Reys +
Prakhar Pandey
Robert de Vries +
Rushabh Vasani
Simon Hawkins
Stijn Van Hoey
Terji Petersen
Tom Augspurger
William Ayd
alimcmaster1
gfyoun

jbrockmendel

5.3.5 What's new in 1.0.1 (February 5, 2020)

These are the changes in 1.0.1. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in *DataFrame* setting values with a slice (e.g. `df[-4:] = 1`) indexing by label instead of position (GH31469)

Fixed regression when indexing a Series or DataFrame indexed by *DatetimeIndex* with a slice containing a `datetime.date` (GH31501)

Fixed regression in *DataFrame.__setitem__* raising an *AttributeError* with a *MultiIndex* and a non-monotonic indexer (GH31449)

Fixed regression in *Series* multiplication when multiplying a numeric *Series* with >10000 elements with a `timedelta`-like scalar (GH31457)

Fixed regression in `.groupby().agg()` raising an *AssertionError* for some reductions like `min` on object-dtype columns (GH31522)

Fixed regression in `.groupby()` aggregations with categorical dtype using Cythonized reduction functions (e.g. `first`) (GH31450)

Fixed regression in *GroupBy.apply()* if called with a function which returned a non- non-scalar object (e.g. a list or numpy array) (GH31441)

Fixed regression in *DataFrame.groupby()* whereby taking the minimum or maximum of a column with period dtype would raise a *TypeError*. (GH31471)

Fixed regression in *DataFrame.groupby()* with an empty DataFrame grouping by a level of a *MultiIndex* (GH31670).

Fixed regression in *DataFrame.apply()* with object dtype and non-reducing function (GH31505)

Fixed regression in `to_datetime()` when parsing non-nanosecond resolution datetimes (GH31491)

Fixed regression in `to_csv()` where specifying an `na_rep` might truncate the values written (GH31447)

Fixed regression in *Categorical* construction with `numpy.str_` categories (GH31499)

Fixed regression in *DataFrame.loc()* and *DataFrame.iloc()* when selecting a row containing a single `datetime64` or `timedelta64` column (GH31649)

Fixed regression where setting `pd.options.display.max_colwidth` was not

accepting negative integer. In addition, this behavior has been deprecated in favor of using None (GH31532)

Fixed regression in objTOJSON.c fix return-type warning (GH31463)

Fixed regression in *qcut()* when passed a nullable integer. (GH31389) Fixed regression in assigning to a *Series* using a nullable integer dtype (GH31446)

Fixed performance regression when indexing a DataFrame or Series with a *MultiIndex* for the index using a list of labels (GH31648)

Fixed regression in *read_csv()* used in file like object RawIOBase is not recognize encoding option

(GH31575)

5.3. Version 1.0

Deprecations

Support for negative integer for `pd.options.display.max_colwidth` is deprecated in favor of using

None (GH31532)

Bug fixes

Datetimelike

Fixed bug in *to_datetime()* raising when `cache=True` and out-of-bound values are present (GH31491)

Numeric

Bug in dtypes being lost in `DataFrame.__invert__` (`~` operator) with mixed dtypes (GH31183) and for extension-array backed Series and DataFrame (GH23087)

Plotting

Plotting tz-aware timeseries no longer gives UserWarning (GH31205)

Interval

Bug in *Series.shift()* with interval dtype raising a `TypeError` when shifting an interval array of integers or datetimes (GH34195)

Contributors

A total of 15 people contributed patches to this release. People with a “+” by

their names contributed a patch for the first time.

Daniel Saxton
Guillaume Lemaitre
Jeff Reback
Joris Van den Bossche
Kaiqi Dong
Marco Gorelli
MeeseeksMachine

Development Team
Sebastián Vanrell +
Tom Augspurger
William Ayd
alimcmaster1
jbrockmendel
paihu +
proost

5.3.6 What's new in 1.0.0 (January 29, 2020)

These are the changes in 1.0.0. See *Release notes* for a full changelog including other versions of .

Note: The 1.0 release removed a lot of functionality that was deprecated in previous releases (see *below* for an overview). It is recommended to first upgrade to 0.25 and to ensure your code is working without warnings, before upgrading to 1.0.

New deprecation policy

Starting with 1.0.0, will adopt a variant of SemVer to version releases. Briefly, Deprecations will be introduced in minor releases (e.g. 1.1.0, 1.2.0, 2.1.0, . . .) Deprecations will be enforced in major releases (e.g. 1.0.0, 2.0.0, 3.0.0, . . .)

API-breaking changes will be made only in major releases (except for experimental features)

See *Version policy* for more.

Enhancements

Using Numba in `rolling.apply` and `expanding.apply`

We've added an `engine` keyword to `apply()` and `apply()` that allows the user to execute the routine using Numba instead of Cython. Using the Numba engine can yield significant performance gains if the apply function can operate on numpy arrays and the data set is larger (1 million rows or greater). For more details, see *rolling apply documentation* (GH28987, GH30936)

Defining custom windows for rolling operations

We've added a `.api.indexers.BaseIndexer()` class that allows users to define how window bounds are created during rolling operations. Users can define their own `get_window_bounds` method on a `.api.indexers.BaseIndexer()` subclass that will generate the start and end indices used for each window during the rolling aggregation. For more details and example usage, see the *custom window rolling documentation*

Converting to markdown

We've added `to_markdown()` for creating a markdown table (GH11052) **In [1]:** `df = pd.DataFrame({"A": [1, 2, 3], "B": [1, 2, 3]}, index=['a', 'a', 'b'])` **In [2]:** `print(df.to_markdown())`

```
| A | B |
|:---|:----|:----|
| a | 1 | 1 |
| a | 2 | 2 |
| b | 3 | 3 |
```

5.3. Version 1.0

Experimental new features

Experimental NA scalar to denote missing values

A new `pd.NA` value (singleton) is introduced to represent scalar missing values. Up to now, used several values to represent missing data: `np.nan` is used for this for float data, `np.nan` or `None` for object-dtype data and `pd.NaT` for datetime-like data. The goal of `pd.NA` is to provide a “missing” indicator that can be used consistently across data types. `pd.NA` is currently used by the nullable integer and boolean data types and the new string data type (GH28095).

Warning: Experimental: the behaviour of pd.NA can still change without warning. For example, creating a Series using the nullable integer dtype:

```
In [3]: s = pd.Series([1, 2, None], dtype="Int64")
```

```
In [4]: s Out[4]:
```

```
1
```

```
2
```

```
<NA>
```

```
Length: 3, dtype: Int64
```

```
In [5]: s[2]
```

```
Out[5]: <NA>
```

Compared to np.nan, pd.NA behaves differently in certain operations. In addition to arithmetic operations, pd.NA also propagates as “missing” or “unknown” in comparison operations:

```
In [6]: np.nan > 1
```

```
Out[6]: False
```

```
In [7]: pd.NA > 1
```

```
Out[7]: <NA>
```

For logical operations, pd.NA follows the rules of the three-valued logic (or *Kleene logic*). For example:

```
In [8]: pd.NA | True
```

```
Out[8]: True
```

For more, see *NA section* in the user guide on missing data.

Dedicated string data type

We’ve added *StringDtype*, an extension type dedicated to string data. Previously, strings were typically stored in object-dtype NumPy arrays. (GH29975) Warning: *StringDtype* is currently considered experimental. The implementation and parts of the API may change without warning. The ‘string’ extension type solves several issues with object-dtype NumPy arrays: You can accidentally store a *mixture* of strings and non-strings in an object dtype array. A *StringArray* can only store strings.

object dtype breaks dtype-specific operations like

`DataFrame.select_dtypes()`. There isn't a clear way to select *just* text while excluding non-text, but still object-dtype columns.

When reading code, the contents of an object dtype array is less clear than string. **In [9]:** `pd.Series(['abc', None, 'def'], dtype=pd.StringDtype())` **Out[9]:**
abc
<NA>
def
Length: 3, dtype: string

You can use the alias "string" as well.

In [10]: `s = pd.Series(['abc', None, 'def'], dtype="string")`

In [11]: `s` **Out[11]:**

abc
<NA>
def
Length: 3, dtype: string

The usual string accessor methods work. Where appropriate, the return type of the Series or columns of a DataFrame will also have string dtype.

In [12]: `s.str.upper()` **Out[12]:**

ABC
<NA>
DEF
Length: 3, dtype: string

In [13]: `s.str.split('b', expand=True).dtypes` **Out[13]:**

string
string
Length: 2, dtype: object

String accessor methods returning integers will return a value with

Int64Dtype **In [14]:** `s.str.count("a")` **Out[14]:**

1
<NA>
0
Length: 3, dtype: Int64

We recommend explicitly using the string data type when working with strings. See *Text data types* for more.

5.3. Version 1.0

Boolean data type with missing values support

We've added *BooleanDtype* / *BooleanArray*, an extension type dedicated to boolean data that can hold missing values. The default bool data type based on a bool-dtype NumPy array, the column can only hold True or False, and not missing values. This new *BooleanArray* can store missing values as well by keeping track of this in a separate mask. (GH29555, GH30095, GH31131)

```
In [15]: pd.Series([True, False, None], dtype=pd.BooleanDtype()) Out[15]:
```

```
True
```

```
False
```

```
<NA>
```

```
Length: 3, dtype: boolean
```

You can use the alias "boolean" as well.

```
In [16]: s = pd.Series([True, False, None], dtype="boolean")
```

```
In [17]: s Out[17]:
```

```
True
```

```
False
```

```
<NA>
```

```
Length: 3, dtype: boolean
```

Method `convert_dtypes` to ease use of supported extension dtypes

In order to encourage use of the extension dtypes `StringDtype`, `BooleanDtype`, `Int64Dtype`, `Int32Dtype`, etc., that support `pd.NA`, the methods

`DataFrame.convert_dtypes()` and `Series.convert_dtypes()` have been introduced. (GH29752) (GH30929)

Example:

```
In [18]: df = pd.DataFrame({'x': ['abc', None, 'def'],
```

```
....: 'y': [1, 2, np.nan], ....: 'z': [True, False, True]})
```

```
....:
```

```
In [19]: df Out[19]:
```

```
x y z
```

```
abc 1.0 True
```

```
None 2.0 False
```

```
def NaN True
[3 rows x 3 columns]
In [20]: df.dtypes Out[20]:
x object y float64 z bool
Length: 3, dtype: object
In [21]: converted = df.convert_dtypes()
In [22]: converted Out[22]:
```

```
x y z
abc 1 True
<NA> 2 False
def <NA> True
[3 rows x 3 columns]
In [23]: converted.dtypes Out[23]:
x string y Int64 z boolean
Length: 3, dtype: object
```

This is especially useful after reading in data using readers such as *read_csv()* and *read_excel()*. See [here](#) for a description.

Other enhancements

DataFrame.to_string() added the *max_colwidth* parameter to control when wide columns are truncated (GH9784)

Added the *na_value* argument to *Series.to_numpy()*, *Index.to_numpy()* and *DataFrame.to_numpy()* to control the value used for missing data (GH30322)

MultiIndex.from_product() infers level names from inputs if not explicitly provided (GH27292)

DataFrame.to_latex() now accepts *caption* and *label* arguments (GH25436)

DataFrames with *nullable integer*, the *new string dtype* and period data type can now be converted to pyarrow ($\geq 0.15.0$), which means that it is supported in writing to the Parquet file format when using the pyarrow engine (GH28368). Full roundtrip to parquet (writing and reading back in with *to_parquet()* / *read_parquet()*) is supported starting with pyarrow ≥ 0.16 (GH20612).

to_parquet() now appropriately handles the *schema* argument for user defined schemas in the pyarrow engine. (GH30270)

DataFrame.to_json() now accepts an indent integer argument to enable pretty printing of JSON output (GH12004)

read_stata() can read Stata 119 dta files. (GH28250)

Implemented `.core.window.Window.var()` and `.core.window.Window.std()` functions (GH26597)

Added encoding argument to *DataFrame.to_string()* for non-ascii text (GH28766) Added encoding argument to *DataFrame.to_html()* for non-ascii text (GH28663) *Styler.background_gradient()* now accepts vmin and vmax arguments (GH12145)

Styler.format() added the na_rep parameter to help format the missing values (GH21527, GH28358)

read_excel() now can read binary Excel (.xlsb) files by passing engine='pyxlsb'. For more details and example usage, see the *Binary Excel files documentation*. Closes GH8540.

The partition_cols argument in *DataFrame.to_parquet()* now accepts a string (GH27117)

5.3. Version 1.0

.read_json() now parses NaN, Infinity and -Infinity (GH12213)

DataFrame constructor preserve ExtensionArray dtype with ExtensionArray (GH11363)

DataFrame.sort_values() and *Series.sort_values()* have gained ignore_index keyword to be able to reset index after sorting (GH30114)

DataFrame.sort_index() and *Series.sort_index()* have gained ignore_index keyword to reset index (GH30114)

DataFrame.drop_duplicates() has gained ignore_index keyword to reset index (GH30114)

Added new writer for exporting Stata dta files in versions 118 and 119, StataWriterUTF8. These file formats support exporting strings containing Unicode characters. Format 119 supports data sets with more than 32,767 variables (GH23573, GH30959)

Series.map() now accepts `collections.abc.Mapping` subclasses as a mapper (GH29733) Added an experimental *attrs* for storing global metadata about a dataset (GH29062) *Timestamp.fromisocalendar()* is now compatible with python 3.8 and above (GH28115) *DataFrame.to_pickle()* and *read_pickle()* now accept URL (GH30163)

Backwards incompatible API changes

Avoid using names from `MultiIndex.levels`

As part of a larger refactor to *MultiIndex* the level names are now stored separately from the levels (GH27242). We recommend using *MultiIndex.names* to access the names, and *Index.set_names()* to update the names.

For backwards compatibility, you can still *access* the names via the levels. **In**

```
[24]: mi = pd.MultiIndex.from_product([[1, 2], ['a', 'b']], names=['x', 'y']) In
```

```
[25]: mi.levels[0].name
```

```
Out[25]: 'x'
```

However, it is no longer possible to *update* the names of the *MultiIndex* via the level. **In** [26]: `mi.levels[0].name = "new name"`

```
-----  
RuntimeErrorTraceback (most recent call last)
```

```
<ipython-input-26-65f4400a0c97> in <module> ----> 1 mi.levels[0].name =  
"new name" ///core/indexes/base.py in name(self, value) 1241 if  
self._no_setting_name:
```

```
1242 # Used in MultiIndex.levels to avoid silently ignoring name , _updates.
```

```
-> 1243 raise RuntimeError(  
"Cannot set name on a level of a MultiIndex. Use "  
"'MultiIndex.set_names' instead."
```

```
RuntimeError: Cannot set name on a level of a MultiIndex. Use  
'MultiIndex.set_names' , → instead.
```

```
In [27]: mi.names
```

```
Out[27]: FrozenList(['x', 'y'])
```

To update, use *MultiIndex.set_names*, which returns a new *MultiIndex*. **In**

```
[28]: mi2 = mi.set_names("new name", level=0)
```

```
In [29]: mi2.names Out[29]: FrozenList(['new name', 'y'])
```

New repr for IntervalArray

.arrays.IntervalArray adopts a new `__repr__` in accordance with other array classes (GH25022) 0.25.x

```
In [1]: pd.arrays.IntervalArray.from_tuples([(0, 1), (2, 3)]) Out[2]:  
IntervalArray([(0, 1], (2, 3]], closed='right', dtype='interval[int64]')
```

1.0.0

```
In [30]: pd.arrays.IntervalArray.from_tuples([(0, 1), (2, 3)]) Out[30]:  
<IntervalArray>  
[(0, 1], (2, 3]]  
Length: 2, closed: right, dtype: interval[int64]
```

DataFrame.rename now only accepts one positional argument

DataFrame.rename() would previously accept positional arguments that would lead to ambiguous or undefined behavior. From 1.0, only the very first argument, which maps labels to their new names along the default axis, is allowed to be passed by position (GH29136). 0.25.x

```
>>> df = pd.DataFrame([[1]])  
>>> df.rename({0: 1}, {0: 2}) FutureWarning: ...Use named arguments to  
resolve ambiguity...  
2  
1 1
```

1.0.0

```
>>> df.rename({0: 1}, {0: 2}) Traceback (most recent call last):
```

```
...
```

```
TypeError: rename() takes from 1 to 2 positional arguments but 3 were given
```

Note that errors will now be raised when conflicting or potentially ambiguous arguments are provided. 0.25.x

```
>>> df.rename({0: 1}, index={0: 2})  
0  
1  
>>> df.rename(mapper={0: 1}, index={0: 2})  
0
```

1

1.0.0

```
>>> df.rename({0: 1}, index={0: 2}) Traceback (most recent call last): ...
TypeError: Cannot specify both 'mapper' and any of 'index' or 'columns' >>>
df.rename(mapper={0: 1}, index={0: 2}) Traceback (most recent call last): ...
TypeError: Cannot specify both 'mapper' and any of 'index' or 'columns'
```

You can still change the axis along which the first positional argument is applied by supplying the axis keyword argument.

In [31]: df.rename({0: 1}) **Out[31]:**

0

1 1

[1 rows x 1 columns]

In [32]: df.rename({0: 1}, axis=1) **Out[32]:**

1

0 1

[1 rows x 1 columns]

If you would like to update both the index and column labels, be sure to use the respective keywords.

In [33]: df.rename(index={0: 1}, columns={0: 2}) **Out[33]:**

2

1 1

[1 rows x 1 columns]

Extended verbose info output for DataFrame

DataFrame.info() now shows line numbers for the columns summary

(GH17304) 0.25.x >>> df = pd.DataFrame({"int_col": [1, 2, 3],

... "text_col": ["a", "b", "c"],

... "float_col": [0.0, 0.1, 0.2]})

>>> df.info(verbose=**True**)

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 3 entries, 0 to 2

Data columns (total 3 columns): int_col 3 non-null int64 text_col 3 non-null
object float_col 3 non-null float64 dtypes: float64(1), int64(1), object(1)
memory usage: 152.0+ bytes

1.0.0

```
In [34]: df = pd.DataFrame({"int_col": [1, 2, 3],  
.....: "text_col": ["a", "b", "c"], .....: "float_col": [0.0, 0.1, 0.2]})  
.....:
```

```
In [35]: df.info(verbose=True)  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 3 entries, 0 to 2  
Data columns (total 3 columns):  
# Column Non-Null Count Dtype
```

```
--- -----  
int_col 3 non-null int64  
text_col 3 non-null object
```

```
float_col 3 non-null float64 dtypes: float64(1), int64(1), object(1) memory  
usage: 200.0+ bytes
```

.array() inference changes

.array() now infers ' new extension types in several cases (GH29791): String data (including missing values) now returns a *arrays.StringArray*. Integer data (including missing values) now returns a *arrays.IntegerArray*.

Boolean data (including missing values) now returns the new *arrays.BooleanArray* 0.25.x

```
>>> pd.array(["a", None])  
<Array>  
['a', None]  
Length: 2, dtype: object  
>>> pd.array([1, None])  
<Array>  
[1, None] Length: 2, dtype: object
```

1.0.0

```
In [36]: pd.array(["a", None]) Out[36]:
```

```
<StringArray>
```

```
['a', <NA>]
```

```
Length: 2, dtype: string
```

```
In [37]: pd.array([1, None]) Out[37]:
```

```
<IntegerArray>
```

```
[1, <NA>]
```

```
Length: 2, dtype: Int64
```

As a reminder, you can specify the dtype to disable all inference.

arrays.IntegerArray now uses .NA

arrays.IntegerArray now uses .NA rather than numpy.nan as its missing value marker (GH29964). 0.25.x

```
>>> a = pd.array([1, 2, None], dtype="Int64")
```

```
>>> a
```

```
<IntegerArray>
```

```
[1, 2, NaN]
```

```
Length: 3, dtype: Int64
```

```
>>> a[2] nan
```

1.0.0

```
In [38]: a = pd.array([1, 2, None], dtype="Int64")
```

```
In [39]: a Out[39]:
```

```
<IntegerArray>
```

```
[1, 2, <NA>]
```

```
Length: 3, dtype: Int64
```

```
In [40]: a[2]
```

```
Out[40]: <NA>
```

This has a few API-breaking consequences.

Converting to a NumPy ndarray

When converting to a NumPy array missing values will be pd.NA, which cannot be converted to a float. So calling np.asarray(integer_array, dtype="float") will now raise.

0.25.x

```
>>> np.asarray(a, dtype="float") array([ 1., 2., nan])
```

1.0.0

```
In [41]: np.asarray(a, dtype="float")
```

```
-----ValueError
```

```
Traceback (most recent call last) <ipython-input-41-4578f04da2b3> in
```

```
<module>
```

```
----> 1 np.asarray(a, dtype="float")
```

```
/opt/conda/envs//lib/python3.8/site-packages/numpy/core/_asarray.py in  
, _asarray(a, dtype, order, like)
```

```
100 return _asarray_with_like(a, dtype=dtype, order=order, like=like) 101
```

```
--> 102 return array(a, dtype, copy=False, order=order)
```

```
103
```

```
104
```

```
///core/arrays/masked.py in __array__(self, dtype)
```

```
We return an object array here to preserve our scalar values
```

```
"""
```

```
--> 239 return self.to_numpy(dtype=dtype)
```

```
240
```

```
241 def __arrow_array__(self, type=None):
```

```
///core/arrays/masked.py in to_numpy(self, dtype, copy, na_value) and  
na_value is libmissing.NA
```

```
):
```

```
--> 220 raise ValueError(
```

```
f"cannot convert to '{dtype}'-dtype NumPy array "
```

```
"with missing values. Specify an appropriate 'na_value' "
```

```
ValueError: cannot convert to 'float64'-dtype NumPy array with missing  
values. , _ Specify an appropriate 'na_value' for this dtype.
```

```
Use arrays.IntegerArray.to_numpy() with an explicit na_value instead. In
```

```
[42]: a.to_numpy(dtype="float", na_value=np.nan)
```

```
Out[42]: array([ 1., 2., nan])
```

```
Reductions can return ``pd.NA``
```

When performing a reduction such as a sum with skipna=False, the result will now be pd.NA instead of np.nan in presence of missing values (GH30958).

```
0.25.x >>> pd.Series(a).sum(skipna=False) nan
```

```
1.0.0
```

```
In [43]: pd.Series(a).sum(skipna=False) Out[43]: <NA>
```

```
value_counts returns a nullable integer dtype
```

Series.value_counts() with a nullable integer dtype now returns a nullable

integer dtype for the values. 0.25.x

```
>>> pd.Series([2, 1, 1, None], dtype="Int64").value_counts().dtype  
dtype('int64') 1.0.0
```

```
In [44]: pd.Series([2, 1, 1, None], dtype="Int64").value_counts().dtype  
Out[44]: Int64Dtype()
```

See *Experimental NA scalar to denote missing values* for more on the differences between .NA and numpy. nan. **arrays.IntegerArray comparisons return arrays.BooleanArray**

Comparison operations on a *arrays.IntegerArray* now returns a *arrays.BooleanArray* rather than a NumPy array (GH29964). 0.25.x

```
>>> a = pd.array([1, 2, None], dtype="Int64")  
>>> a  
<IntegerArray>  
[1, 2, NaN]  
Length: 3, dtype: Int64  
>>> a > 1  
array([False, True, False])
```

1.0.0

```
In [45]: a = pd.array([1, 2, None], dtype="Int64")
```

```
In [46]: a > 1
```

```
Out[46]:
```

```
<BooleanArray>  
[False, True, <NA>]  
Length: 3, dtype: boolean
```

Note that missing values now propagate, rather than always comparing unequal like numpy.nan. See *Experimental NA scalar to denote missing values* for more.

By default Categorical.min() now returns the minimum instead of np.nan When *Categorical* contains np.nan, *Categorical.min()* no longer return np.nan by default (skipna=True) (GH25303) 0.25.x

```
In [1]: pd.Categorical([1, 2, np.nan], ordered=True).min() Out[1]: nan
```


1.0.0

In [47]: pd.Categorical([1, 2, np.nan], ordered=**True**).min()

Out[47]: 1

Default dtype of empty .Series

Initialising an empty *.Series* without specifying a dtype will raise a `DeprecationWarning` now (GH17261). The default dtype will change from `float64` to `object` in future releases so that it is consistent with the behaviour of *DataFrame* and *Index*. 1.0.0

In [1]: pd.Series() **Out[2]:**

`DeprecationWarning: The default dtype for empty Series will be 'object' instead of ,→ 'float64' in a future version. Specify a dtype explicitly to silence this warning. Series([], dtype: float64)`

Result dtype inference changes for resample operations

The rules for the result dtype in *DataFrame.resample()* aggregations have changed for extension types (GH31359). Previously, would attempt to convert the result back to the original dtype, falling back to the usual inference rules if that was not possible. Now, will only return a result of the original dtype if the scalar values in the result are instances of the extension dtype's scalar type.

In [48]: df = pd.DataFrame({"A": ['a', 'b']}, dtype='category', ...:
index=pd.date_range('2000', periods=2))

....:

In [49]: df **Out[49]:**

```
A
2000-01-01 a
2000-01-02 b
[2 rows x 1 columns]
```

0.25.x

```
>>> df.resample("2D").agg(lambda x: 'a').A.dtype
CategoricalDtype(categories=['a', 'b'], ordered=False)
```

1.0.0

```
In [50]: df.resample("2D").agg(lambda x: 'a').A.dtype  
Out[50]: dtype('O')
```

This fixes an inconsistency between resample and groupby. This also fixes a potential bug, where the values of the result might change depending on how the results are cast back to the original dtype. 0.25.x

```
>>> df.resample("2D").agg(lambda x: 'c')  
A  
0 NaN
```

1.0.0

```
In [51]: df.resample("2D").agg(lambda x: 'c') Out[51]:
```

```
A  
2000-01-01 c  
[1 rows x 1 columns]
```

Increased minimum version for Python

1.0.0 supports Python 3.6.1 and higher (GH29212).

Increased minimum versions for dependencies

Some minimum supported versions of dependencies were updated (GH29766, GH29723). If installed, we now require: Package

numpy pytz

python
dateutil

bottleneck numexpr Minimum Version

1.13.3
2015.4

2.6.1
X

1.2.1
2.6.2

Required Changed

X

X pytest 4.0.2
(dev)

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of . Optional libraries below the lowest tested version may still work, but are not considered supported.

Package Minimum Changed Version

beautifulsoup4 4.6.0 fastparquet 0.3.2 X gcsfs 0.2.2 lxml 3.8.0 matplotlib 2.2.2 numba 0.46.0 X openpyxl 2.5.7 X pyarrow 0.13.0 X pymysql 0.7.1 pytables 3.4.2 s3fs 0.3.0 X scipy 0.19.0 sqlalchemy 1.1.4 xarray 0.8.2 xlrd 1.1.0 xlsxwriter 0.9.8 xlwt 1.2.0 See *Dependencies and Optional dependencies* for more.

Build changes

has added a pyproject.toml file and will no longer include cythonized files in the source distribution uploaded to PyPI (GH28341, GH20775). If you're installing a built distribution (wheel) or via conda, this shouldn't have any effect on you. If you're building from source, you should no longer need to install Cython into your build environment before calling pip install .

Other API changes

core.groupby.GroupBy.transform now raises on invalid operation names (GH27489)

.api.types.infer_dtype() will now return "integer-na" for integer and np.nan mix (GH27283)

MultiIndex.from_arrays() will no longer infer names from arrays if names=None is explicitly provided (GH27292)

In order to improve tab-completion, does not include most deprecated attributes when introspecting a object using dir (e.g. dir(df)). To see which attributes are excluded, see an object's _deprecations attribute, for example

`pd.DataFrame._deprecations` (GH28805).

The returned dtype of `unique()` now matches the input dtype. (GH27874)

Changed the default configuration value for `options.matplotlib.register_converters` from `True` to `"auto"` (GH18720). Now, custom formatters will only be applied to plots created by , through `plot()`. Previously, ' formatters would be applied to all plots created *after* a `plot()`. See *units registration* for more.

`Series.dropna()` has dropped its `**kwargs` argument in favor of a single `how` parameter. Supplying anything else than `how` to `**kwargs` raised a `TypeError` previously (GH29388)

When testing , the new minimum required version of `pytest` is 5.0.1 (GH29664) `Series.str.__iter__()` was deprecated and will be removed in future releases (GH28277). Added `<NA>` to the list of default NA values for `read_csv()` (GH30821)

Documentation improvements

Added new section on *Scaling to large datasets* (GH28315).

Added sub-section on *Query MultiIndex* for HDF5 datasets (GH28791).

Deprecations

`Series.item()` and `Index.item()` have been `_undeprecated_` (GH29250)

`Index.set_value` has been deprecated. For a given index `idx`, array `arr`, value in `idx` of `idx_val` and a new value of `val`, `idx.set_value(arr, idx_val, val)` is equivalent to `arr[idx.get_loc(idx_val)] = val`, which should be used instead (GH28621).

`is_extension_type()` is deprecated, `is_extension_array_dtype()` should be used instead (GH29457)

`eval()` keyword argument `"truediv"` is deprecated and will be removed in a future version (GH29812)

5.3. Version 1.0

`DateOffset.isAnchored()` and `DatetimeOffset.onOffset()` are deprecated and will be removed in a future version, use `DateOffset.is_anchored()` and `DateOffset.is_on_offset()` instead (GH30340)

`.tseries.frequencies.get_offset` is deprecated and will be removed in a future version, use `.tseries.frequencies.to_offset` instead (GH4205)

`Categorical.take_nd()` and `CategoricalIndex.take_nd()` are deprecated, use `Categorical.take()` and `CategoricalIndex.take()` instead (GH27745)

The parameter `numeric_only` of `Categorical.min()` and `Categorical.max()` is deprecated and replaced with `skipna` (GH25303)

The parameter `label` in `lreshape()` has been deprecated and will be removed in a future version (GH29742)

`.core.index` has been deprecated and will be removed in a future version, the public classes are available in the top-level namespace (GH19711)

`.json_normalize()` is now exposed in the top-level namespace. Usage of `json_normalize` as `.io.json.json_normalize` is now deprecated and it is recommended to use `json_normalize` as `.json_normalize()` instead (GH27586).

The `numpy` argument of `.read_json()` is deprecated (GH28512).

`DataFrame.to_stata()`, `DataFrame.to_feather()`, and `DataFrame.to_parquet()` argument “`fname`” is deprecated, use “`path`” instead (GH23574)

The deprecated internal attributes `_start`, `_stop` and `_step` of `RangeIndex` now raise a `FutureWarning` instead of a `DeprecationWarning` (GH26581)

The `.util.testing` module has been deprecated. Use the public API in `.testing` documented at *Testing functions* (GH16232).

`.SparseArray` has been deprecated. Use `.arrays.SparseArray` (`arrays.SparseArray`) instead. (GH30642)

The parameter `is_copy` of `Series.take()` and `DataFrame.take()` has been deprecated and will be removed in a future version. (GH27357)

Support for multi-dimensional indexing (e.g. `index[:, None]`) on a `Index` is deprecated and will be removed in a future version, convert to a `numpy` array before indexing instead (GH30588)

The `.np` submodule is now deprecated. Import `numpy` directly instead (GH30296)

The `.datetime` class is now deprecated. Import from `datetime` instead (GH30610) `diff` will raise a `TypeError` rather than implicitly losing the `dtype` of extension types in the future. Convert to the correct `dtype` before calling `diff` instead (GH31025) Selecting Columns from a Grouped `DataFrame`

When selecting columns from a `DataFrameGroupBy` object, passing individual keys (or a tuple of keys) inside single brackets is deprecated, a list of items should be used instead. (GH23566) For example:

```
df = pd.DataFrame({
    "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
    "B": np.random.randn(8),
    "C": np.random.randn(8),
}) g = df.groupby('A')
# single key, returns SeriesGroupBy
(continues on next page)
(continued from previous page)
```

```
g['B']
# tuple of single key, returns SeriesGroupBy g[('B,')]
# tuple of multiple keys, returns DataFrameGroupBy, raises FutureWarning
g[('B', 'C')] # multiple keys passed directly, returns DataFrameGroupBy,
raises FutureWarning # (implicitly converts the passed strings into a single
tuple) g['B', 'C']
# proper way, returns DataFrameGroupBy g[['B', 'C']]
```

Removal of prior version deprecations/changes

Removed `SparseSeries` and `SparseDataFrame`

`SparseSeries`, `SparseDataFrame` and the `DataFrame.to_sparse` method have been removed (GH28425). We recommend using a `Series` or `DataFrame` with sparse values instead. See *Migrating* for help with migrating existing code.

Matplotlib unit registration

Previously, would register converters with matplotlib as a side effect of importing (GH18720). This changed the output of plots made via matplotlib plots after was imported, even if you were using matplotlib directly rather than `plot()`.

To use formatters with a matplotlib plot, specify

```
>>> import as pd
>>> pd.options.plotting.matplotlib.register_converters = True
```

Note that plots created by *DataFrame.plot()* and *Series.plot()* do register the converters automatically. The only behavior change is when plotting a date-like object via `matplotlib.pyplot.plot` or `matplotlib.Axes.plot`. See *Custom formatters for timeseries plots* for more.

Other removals

Removed the previously deprecated keyword “index” from *read_stata()*, *StataReader*, and *StataReader.read()*, use “index_col” instead (GH17328)

Removed *StataReader.data* method, use *StataReader.read()* instead (GH9493)

Removed *.plotting._matplotlib.tsplot*, use *Series.plot()* instead (GH19980)

.tseries.converter.register has been moved to *.plotting*.

register_matplotlib_converters() (GH18307)

Series.plot() no longer accepts positional arguments, pass keyword arguments instead (GH30003)

DataFrame.hist() and *Series.hist()* no longer allows `figsize="default"`, specify figure size by passing a tuple instead (GH30003)

Floordiv of integer-dtyped array by *Timedelta* now raises *TypeError* (GH21036)

TimedeltaIndex and *DatetimeIndex* no longer accept non-nanosecond dtype strings like “timedelta64” or “datetime64”, use “timedelta64[ns]” and “datetime64[ns]” instead (GH24806)

5.3. Version 1.0

Changed the default “skipna” argument in *.api.types.infer_dtype()* from False to True (GH24050)

Removed *Series.ix* and *DataFrame.ix* (GH26438)

Removed *Index.summary* (GH18217)

Removed the previously deprecated keyword “fastpath” from the *Index* constructor (GH23110)

Removed *Series.get_value*, *Series.set_value*, *DataFrame.get_value*, *DataFrame.set_value* (GH17739)

Removed *Series.compound* and *DataFrame.compound* (GH26405)

Changed the default “inplace” argument in *DataFrame.set_index()* and

Series.set_axis() from None to False (GH27600)

Removed *Series.cat.categorical*, *Series.cat.index*, *Series.cat.name* (GH24751)

Removed the previously deprecated keyword “box” from *to_datetime()* and *to_timedelta()*; in addition these now always returns *DatetimeIndex*, *TimedeltaIndex*, *Index*, *Series*, or *DataFrame* (GH24486)

to_timedelta(), *Timedelta*, and *TimedeltaIndex* no longer allow “M”, “y”, or “Y” for the “unit” argument (GH23264)

Removed the previously deprecated keyword “time_rule” from (non -public) *offsets.generate_range*, which has been moved to *core.arrays._ranges.generate_range()* (GH24157)

DataFrame.loc() or *Series.loc()* with listlike indexers and missing labels will no longer reindex (GH17295)

DataFrame.to_excel() and *Series.to_excel()* with non-existent columns will no longer reindex (GH17295)

Removed the previously deprecated keyword “join_axes” from *concat()*; use *reindex_like* on the result instead (GH22318)

Removed the previously deprecated keyword “by” from *DataFrame.sort_index()*, use *DataFrame.sort_values()* instead (GH10726)

Removed support for nested renaming in *DataFrame.aggregate()*,

Series.aggregate() , *core.groupby.DataFrameGroupBy.aggregate()*,
core.groupby.SeriesGroupBy.aggregate(),
core.window.rolling.Rolling.aggregate() (GH18529)

Passing *datetime64* data to *TimedeltaIndex* or *timedelta64* data to *DatetimeIndex* now raises *TypeError* (GH23539, GH23937)

Passing *int64* values to *DatetimeIndex* and a *timezone* now interprets the values as *nanosecond* timestamps in *UTC*, not wall times in the given *timezone* (GH24559) A tuple passed to *DataFrame.groupby()* is now exclusively treated as a single key (GH18314)

Removed *Index.contains*, use *key* in *index* instead (GH30103)

Addition and subtraction of *int* or *integer-arrays* is no longer allowed in *Timestamp*, *DatetimeIndex*, *TimedeltaIndex*, use *obj + n * obj.freq* instead of *obj + n* (GH22535)

Removed `Series.ptp` (GH21614)

Removed `Series.from_array` (GH18258)

Removed `DataFrame.from_items` (GH18458)

Removed `DataFrame.as_matrix`, `Series.as_matrix` (GH18458) Removed `Series.asobject` (GH18477)

Removed `DataFrame.as_blocks`, `Series.as_blocks`, `DataFrame.blocks`, `Series.blocks` (GH17656)

`.Series.str.cat()` now defaults to aligning others, using `join='left'` (GH27611)

`.Series.str.cat()` does not accept list-likes *within* list-likes anymore (GH27611)

`Series.where()` with Categorical dtype (or `DataFrame.where()` with Categorical column) no longer allows setting new categories (GH24114)

Removed the previously deprecated keywords “start”, “end”, and “periods” from the `DatetimeIndex`,

`TimedeltaIndex`, and `PeriodIndex` constructors; use `date_range()`, `timedelta_range()`, and `period_range()` instead (GH23919)

Removed the previously deprecated keyword “verify_integrity” from the `DatetimeIndex` and `TimedeltaIndex` constructors (GH23919) Removed the previously deprecated keyword “fastpath” from `.core.internals.blocks.make_block` (GH19265)

Removed the previously deprecated keyword “dtype” from `Block.make_block_same_class()` (GH19434)

Removed `ExtensionArray._formatting_values`. Use `ExtensionArray._formatter` instead. (GH23601)

Removed `MultiIndex.to_hierarchical` (GH21613)

Removed `MultiIndex.labels`, use `MultiIndex.codes` instead (GH23752)

Removed the previously deprecated keyword “labels” from the `MultiIndex` constructor, use “codes” instead (GH23752)

Removed `MultiIndex.set_labels`, use `MultiIndex.set_codes()` instead (GH23752) Removed the previously deprecated keyword “labels” from `MultiIndex.set_codes()`, `MultiIndex.copy()`, `MultiIndex.drop()`, use “codes” instead (GH23752)

Removed support for legacy HDF5 formats (GH29787)

Passing a dtype alias (e.g. 'datetime64[ns, UTC]') to *DatetimeTZDtype* is no longer allowed, use *DatetimeTZDtype.construct_from_string()* instead (GH23990) Removed the previously deprecated keyword "skip_footer" from *read_excel()*; use "skipfooter" instead (GH18836)

read_excel() no longer allows an integer value for the parameter usecols, instead pass a list of integers from 0 to usecols inclusive (GH23635) Removed the previously deprecated keyword "convert_datetime64" from *DataFrame.to_records()* (GH18902)

Removed *IntervalIndex.from_intervals* in favor of the *IntervalIndex* constructor (GH19263)

Changed the default "keep_tz" argument in *DatetimeIndex.to_series()* from None to True (GH23739)

Removed *api.types.is_period* and *api.types.is_datetimetz* (GH23917)

Ability to read pickles containing *Categorical* instances created with pre-0.16 version of has been removed (GH27538)

Removed *.series.plotting.tsplot* (GH18627)

5.3. Version 1.0

Removed the previously deprecated keywords "reduce" and "broadcast" from *DataFrame.apply()* (GH18577)

Removed the previously deprecated *assert_raises_regex* function in *._testing* (GH29174)

Removed the previously deprecated *FrozenNDArray* class in *.core.indexes.frozen* (GH29335)

Removed the previously deprecated keyword "nthreads" from *read_feather()*, use "use_threads" instead (GH23053)

Removed *Index.is_lexsorted_for_tuple* (GH29305)

Removed support for nested renaming in *DataFrame.aggregate()*, *Series.aggregate()*, *core.groupby.DataFrameGroupBy.aggregate()*, *core.groupby.SeriesGroupBy*.

aggregate() , *core.window.rolling.Rolling.aggregate()* (GH29608)

Removed *Series.valid*; use *Series.dropna()* instead (GH18800)

Removed *DataFrame.is_copy*, *Series.is_copy* (GH18812)

Removed *DataFrame.get_ftype_counts*, *Series.get_ftype_counts* (GH18243)

Removed `DataFrame.ftypes`, `Series.ftypes`, `Series.ftype` (GH26744)
Removed `Index.get_duplicates`, use `idx[idx.duplicated()].unique()` instead (GH20239)

Removed `Series.clip_upper`, `Series.clip_lower`, `DataFrame.clip_upper`, `DataFrame.clip_lower` (GH24203)
Removed the ability to alter `DatetimeIndex.freq`, `TimedeltaIndex.freq`, or `PeriodIndex.freq` (GH20772)

Removed `DatetimeIndex.offset` (GH20730)
Removed `DatetimeIndex.asobject`, `TimedeltaIndex.asobject`, `PeriodIndex.asobject`, use `astype(object)` instead (GH29801)

Removed the previously deprecated keyword “order” from `factorize()` (GH19751) Removed the previously deprecated keyword “encoding” from `read_stata()` and `DataFrame.to_stata()` (GH21400)
Changed the default “sort” argument in `concat()` from `None` to `False` (GH20613) Removed the previously deprecated keyword “raise_conflict” from `DataFrame.update()`, use “errors” instead (GH23585)
Removed the previously deprecated keyword “n” from `DatetimeIndex.shift()`, `TimedeltaIndex.shift()`, `PeriodIndex.shift()`, use “periods” instead (GH22458) Removed the previously deprecated keywords “how”, “fill_method”, and “limit” from `DataFrame.resample()` (GH30139)
Passing an integer to `Series.fillna()` or `DataFrame.fillna()` with `timedelta64[ns]` dtype now raises `TypeError` (GH24694)

Passing multiple axes to `DataFrame.dropna()` is no longer supported (GH20995) Removed `Series.nonzero`, use `to_numpy().nonzero()` instead (GH24048)

Passing floating dtype codes to `Categorical.from_codes()` is no longer supported, pass codes. `astype(np.int64)` instead (GH21775)
Removed the previously deprecated keyword “pat” from `Series.str.partition()` and `Series.str`.

`rpartition()`, use “sep” instead (GH23767)
Removed `Series.put` (GH27106)
Removed `Series.real`, `Series.imag` (GH27106)

Removed `Series.to_dense`, `DataFrame.to_dense` (GH26684)
Removed `Index.dtype_str`, use `str(index.dtype)` instead (GH27106)
`Categorical.ravel()` returns a *Categorical* instead of a ndarray (GH27199)

The ‘outer’ method on Numpy ufuncs, e.g. `np.subtract.outer` operating on *Series* objects is no longer supported, and will raise `NotImplementedError` (GH27198) Removed `Series.get_dtype_counts` and `DataFrame.get_dtype_counts` (GH27145)

Changed the default “fill_value” argument in `Categorical.take()` from `True` to `False` (GH20841)

Changed the default value for the `raw` argument in `Series.rolling().apply()`, `DataFrame.rolling().apply()`, `Series.expanding().apply()`, and `DataFrame.expanding().apply()` from `None` to `False` (GH20584)

Removed deprecated behavior of `Series.argmin()` and `Series.argmax()`, use `Series.idxmin()` and `Series.idxmax()` for the old behavior (GH16955)

Passing a tz-aware `datetime.datetime` or *Timestamp* into the *Timestamp* constructor with the `tz` argument now raises a `ValueError` (GH23621)

Removed `Series.base`, `Index.base`, `Categorical.base`, `Series.flags`, `Index.flags`, `PeriodArray.flags`, `Series.strides`, `Index.strides`, `Series.itemsize`, `Index.itemsize`, `Series.data`, `Index.data` (GH20721)

Changed `Timedelta.resolution()` to match the behavior of the standard library `datetime`.

`timedelta.resolution`, for the old behavior, use `Timedelta.resolution_string()` (GH26839) Removed `Timestamp.weekday_name`,

`DatetimeIndex.weekday_name`, and `Series.dt.weekday_name` (GH18164)

Removed the previously deprecated keyword “errors” in `Timestamp.tz_localize()`, `DatetimeIndex.tz_localize()`, and `Series.tz_localize()` (GH22644)

Changed the default “ordered” argument in `CategoricalDtype` from `None` to `False` (GH26336)

`Series.set_axis()` and `DataFrame.set_axis()` now require “labels” as the first argument and “axis” as an optional named parameter (GH30089)

Removed `to_msgpack`, `read_msgpack`, `DataFrame.to_msgpack`, `Series.to_msgpack` (GH27103)

Removed `Series.compress` (GH21930)

Removed the previously deprecated keyword “fill_value” from `Categorical.fillna()`, use “value” instead (GH19269)

Removed the previously deprecated keyword “data” from `andrews_curves()`, use “frame” instead (GH6956)

Removed the previously deprecated keyword “data” from `parallel_coordinates()`, use “frame” instead (GH6956)

Removed the previously deprecated keyword “colors” from `parallel_coordinates()`, use “color” instead (GH6956)

Removed the previously deprecated keywords “verbose” and “private_key” from `read_gbq()` (GH30200)

Calling `np.array` and `np.asarray` on tz-aware *Series* and *DatetimeIndex* will now return an object array of tz-aware *Timestamp* (GH24596)

5.3. Version 1.0

Performance improvements

Performance improvement in *DataFrame* arithmetic and comparison operations with scalars (GH24990, GH29853)

Performance improvement in indexing with a non-unique *IntervalIndex* (GH27489) Performance improvement in `MultiIndex.is_monotonic` (GH27495)

Performance improvement in `cut()` when bins is an *IntervalIndex* (GH27668)

Performance improvement when initializing a *DataFrame* using a range (GH30171) Performance improvement in `DataFrame.corr()` when method is “spearman” (GH28139)

Performance improvement in `DataFrame.replace()` when provided a list of values to replace (GH28099)

Performance improvement in `DataFrame.select_dtypes()` by using vectorization instead of iterating over a loop (GH28317)

Performance improvement in `Categorical.searchsorted()` and `CategoricalIndex.searchsorted()` (GH28795)

Performance improvement when comparing a *Categorical* with a scalar and the scalar is not found in the categories (GH29750)

Performance improvement when checking if values in a *Categorical* are

equal, equal or larger or larger than a given scalar. The improvement is not present if checking if the *Categorical* is less than or less than or equal than the scalar (GH29820)

Performance improvement in *Index.equals()* and *MultiIndex.equals()* (GH29134) Performance improvement in *infer_dtype()* when *skipna* is True (GH28814) **Bug fixes**

Categorical

Added test to assert the *fillna()* raises the correct *ValueError* message when the value isn't a value from categories (GH13628)

Bug in *Categorical.astype()* where NaN values were handled incorrectly when casting to int (GH28406)

DataFrame.reindex() with a *CategoricalIndex* would fail when the targets contained duplicates, and wouldn't fail if the source contained duplicates (GH28107)

Bug in *Categorical.astype()* not allowing for casting to extension dtypes (GH28668) Bug where *merge()* was unable to join on categorical and extension dtype columns (GH28668)

Categorical.searchsorted() and *CategoricalIndex.searchsorted()* now work on unordered categoricals also (GH21667)

Added test to assert roundtripping to parquet with *DataFrame.to_parquet()* or *read_parquet()* will preserve Categorical dtypes for string types (GH27955) Changed the error message in *Categorical.remove_categories()* to always show the invalid removals as a set (GH28669)

Using date accessors on a categorical dtyped *Series* of datetimes was not returning an object of the same type as if one used the *str.()* / *dt.()* on a *Series* of that type. E.g. when accessing *Series.dt.tz_localize()* on a *Categorical* with duplicate entries, the accessor was skipping duplicates (GH27952)

Bug in *DataFrame.replace()* and *Series.replace()* that would give incorrect results on categorical data (GH26988)

Bug where calling *Categorical.min()* or *Categorical.max()* on an empty *Categorical* would raise a numpy exception (GH30227)

The following methods now also correctly output values for unobserved categories when called through `groupby(..., observed=False)` (GH17605) * `core.groupby.SeriesGroupBy.count()` * `core.groupby.SeriesGroupBy.size()` * `core.groupby.SeriesGroupBy.nunique()` * `core.groupby.SeriesGroupBy.nth()`

Datetimelike

Bug in `Series.__setitem__()` incorrectly casting `np.timedelta64("NaT")` to `np.datetime64("NaT")` when inserting into a *Series* with `datetime64` dtype (GH27311) Bug in `Series.dt()` property lookups when the underlying data is read-only (GH27529) Bug in `HDFStore.__getitem__` incorrectly reading `tz` attribute created in Python 2 (GH26443)

Bug in `to_datetime()` where passing arrays of malformed str with `errors="coerce"` could incorrectly lead to raising `ValueError` (GH28299)

Bug in `core.groupby.SeriesGroupBy.nunique()` where `NaT` values were interfering with the count of unique values (GH27951)

Bug in *Timestamp* subtraction when subtracting a *Timestamp* from a `np.datetime64` object incorrectly raising `TypeError` (GH28286)

Addition and subtraction of integer or integer-dtype arrays with *Timestamp* will now raise `NullFrequencyError` instead of `ValueError` (GH28268)

Bug in *Series* and *DataFrame* with integer dtype failing to raise `TypeError` when adding or subtracting a `np.datetime64` object (GH28080)

Bug in `Series.astype()`, `Index.astype()`, and `DataFrame.astype()` failing to handle `NaT` when casting to an integer dtype (GH28492)

Bug in *Week* with `weekday` incorrectly raising `AttributeError` instead of `TypeError` when adding or subtracting an invalid type (GH28530)

Bug in *DataFrame* arithmetic operations when operating with a *Series* with dtype `'timedelta64[ns]'` (GH28049)

Bug in `core.groupby.generic.SeriesGroupBy.apply()` raising `ValueError` when a column in the original *DataFrame* is a `datetime` and the column labels are not standard integers (GH28247)

Bug in `._config.localization.get_locales()` where the `locales -a` encodes the `locales` list as `windows-1252` (GH23638, GH24760, GH27368)

Bug in `Series.var()` failing to raise `TypeError` when called with `timedelta64[ns]` dtype (GH28289)

Bug in *DatetimeIndex.strftime()* and *Series.dt.strftime()* where NaT was converted to the string 'NaT' instead of np.nan (GH29578)

5.3. Version 1.0

Bug in masking datetime-like arrays with a boolean mask of an incorrect length not raising an *IndexError* (GH30308)

Bug in *Timestamp.resolution* being a property instead of a class attribute (GH29910) Bug in *.to_datetime()* when called with None raising *TypeError* instead of returning NaT (GH30011)

Bug in *.to_datetime()* failing for dequeues when using *cache=True* (the default) (GH29403)

Bug in *Series.item()* with *datetime64* or *timedelta64* dtype, *DatetimeIndex.item()*, and *TimedeltaIndex.item()* returning an integer instead of a *Timestamp* or *Timedelta* (GH30175)

Bug in *DatetimeIndex* addition when adding a non-optimized *DateOffset* incorrectly dropping timezone information (GH30336)

Bug in *DataFrame.drop()* where attempting to drop non-existent values from a *DatetimeIndex* would yield a confusing error message (GH30399)

Bug in *DataFrame.append()* would remove the timezone-awareness of new data (GH30238)

Bug in *Series.cummin()* and *Series.cummax()* with timezone-aware dtype incorrectly dropping its timezone (GH15553)

Bug in *DatetimeArray*, *TimedeltaArray*, and *PeriodArray* where inplace addition and subtraction did not actually operate inplace (GH24115)

Bug in *.to_datetime()* when called with *Series* storing *IntegerArray* raising *TypeError* instead of returning *Series* (GH30050)

Bug in *date_range()* with custom business hours as *freq* and given number of periods (GH30593)

Bug in *PeriodIndex* comparisons with incorrectly casting integers to *Period* objects, inconsistent with the *Period* comparison behavior (GH30722)

Bug in *DatetimeIndex.insert()* raising a *ValueError* instead of a *TypeError* when trying to insert a timezone-aware *Timestamp* into a timezone-naive *DatetimeIndex*, or vice-versa (GH30806)

Timedelta

Bug in subtracting a *TimedeltaIndex* or *TimedeltaArray* from a *np.datetime64* object (GH29558)

-

Timezones

-

Numeric

Bug in *DataFrame.quantile()* with zero-column *DataFrame* incorrectly raising (GH23925)

DataFrame flex inequality comparisons methods (*DataFrame.lt()*, *DataFrame.le()*, *DataFrame.gt()*, *DataFrame.ge()*) with object-dtype and complex entries failing to raise *TypeError* like their *Series* counterparts (GH28079)

Bug in *DataFrame* logical operations (&, |, ^) not matching *Series* behavior by filling NA values (GH28741)

pandas: powerful Python data analysis toolkit, Release 1.2.3: powerful Python data analysis toolkit, Release 1.2.3

-

-

Bug in *DataFrame.interpolate()* where specifying axis by name references variable before it is assigned (GH29142)

Bug in *Series.var()* not computing the right value with a nullable integer dtype series not passing through *ddof* argument (GH29128)

Improved error message when using *frac* > 1 and *replace* = False (GH27451)

Bug in numeric indexes resulted in it being possible to instantiate an *Int64Index*, *UInt64Index*, or *Float64Index* with an invalid dtype (e.g. datetime-like) (GH29539)

Bug in *UInt64Index* precision loss while constructing from a list with values in the *np.uint64* range (GH29526)

Bug in *NumericIndex* construction that caused indexing to fail when integers in the *np.uint64* range were used (GH28023)

Bug in *NumericIndex* construction that caused *UInt64Index* to be casted to

Float64Index when integers in the `np.uint64` range were used to index a *DataFrame* (GH28279) Bug in *Series.interpolate()* when using `method='index'` with an unsorted index, would previously return incorrect results. (GH21037)

Bug in *DataFrame.round()* where a *DataFrame* with a *CategoricalIndex* of *IntervalIndex* columns would incorrectly raise a `TypeError` (GH30063)

Bug in *Series.pct_change()* and *DataFrame.pct_change()* when there are duplicated indices (GH30463)

Bug in *DataFrame* cumulative operations (e.g. `cumsum`, `cummax`) incorrect casting to object-dtype (GH19296)

Bug in *diff* losing the dtype for extension types (GH30889)

Bug in *DataFrame.diff* raising an `IndexError` when one of the columns was a nullable integer dtype

(GH30967)

Conversion

-

Strings

Calling *Series.str.isalnum()* (and other “ismethods”) on an empty *Series* would return an object dtype instead of `bool` (GH29624)

-

Interval

Bug in *IntervalIndex.get_indexer()* where a *Categorical* or *CategoricalIndex* target would incorrectly raise a `TypeError` (GH30063)

Bug in `.core.dtypes.cast.infer_dtype_from_scalar` where passing `_dtype=True` did not infer *IntervalDtype* (GH30337)

Bug in *Series* constructor where constructing a *Series* from a list of *Interval* objects resulted in object dtype instead of *IntervalDtype* (GH23563)

5.3. Version 1.0

Bug in *IntervalDtype* where the `kind` attribute was incorrectly set as `None` instead of `"O"` (GH30568)

Bug in *IntervalIndex*, *IntervalArray*, and *Series* with interval data where equality comparisons were incorrect (GH24112)

Indexing

Bug in assignment using a reverse slicer (GH26939)

Bug in *DataFrame.explode()* would duplicate frame in the presence of duplicates in the index (GH28010)

Bug in reindexing a *PeriodIndex()* with another type of index that contained a Period (GH28323) (GH28337)

Fix assignment of column via *.loc* with numpy non-ns datetime type (GH27395) Bug in *Float64Index.astype()* where *np.inf* was not handled properly when casting to an integer dtype (GH28475)

Index.union() could fail when the left contained duplicates (GH28257)

Bug when indexing with *.loc* where the index was a *CategoricalIndex* with non-string categories didn't work (GH17569, GH30225)

Index.get_indexer_non_unique() could fail with *TypeError* in some cases, such as when searching for ints in a string index (GH28257)

Bug in *Float64Index.get_loc()* incorrectly raising *TypeError* instead of *KeyError* (GH29189)

Bug in *DataFrame.loc()* with incorrect dtype when setting Categorical value in 1-row *DataFrame* (GH25495)

MultiIndex.get_loc() can't find missing values when input includes missing values Bug in *Series.__setitem__()* incorrectly assigning values with boolean indexer when the length of new data matches the number of True values and new data is not a *Series* or an *np.array* (GH30567)

Bug in indexing with a *PeriodIndex* incorrectly accepting integers representing years, use e.g. *ser.loc[*

"2007"] instead of *ser.loc[2007]* (GH30763)

Missing

-

MultiIndex

Constructor for *MultiIndex* verifies that the given sortorder is compatible with the actual *lexsort_depth* if *verify_integrity* parameter is *True* (the default) (GH28735) *Series* and *MultiIndex* *.drop* with *MultiIndex* raise exception if labels not in given in level (GH8594)

-

IO

read_csv() now accepts binary mode file buffers when using the Python csv engine (GH23779)

Bug in *DataFrame.to_json()* where using a Tuple as a column or index value and using *orient="columns"* or *orient="index"* would produce invalid JSON (GH20500) Improve infinity parsing. *read_csv()* now interprets Infinity, +Infinity, -Infinity as floating point values (GH10065)

Bug in *DataFrame.to_csv()* where values were truncated when the length of *na_rep* was shorter than the text input data. (GH25099)

Bug in *DataFrame.to_string()* where values were truncated using display options instead of outputting the full content (GH9784)

Bug in *DataFrame.to_json()* where a datetime column label would not be written out in ISO format with *orient="table"* (GH28130)

Bug in *DataFrame.to_parquet()* where writing to GCS would fail with *engine='fastparquet'* if the file did not already exist (GH28326)

Bug in *read_hdf()* closing stores that it didn't open when Exceptions are raised (GH28699)

Bug in *DataFrame.read_json()* where using *orient="index"* would not maintain the order Bug in *DataFrame.to_html()* where the length of the *formatters* argument was not verified (GH28469)

Bug in *DataFrame.read_excel()* with *engine='ods'* when *sheet_name* argument references a non-existent sheet (GH27676)

Bug in *.io.formats.style.Styler()* formatting for floating values not displaying decimals correctly (GH13257)

Bug in *DataFrame.to_html()* when using *formatters=<list>* and *max_cols* together. (GH25955)

Bug in *Styler.background_gradient()* not able to work with dtype *Int64* (GH28869) Bug in *DataFrame.to_clipboard()* which did not work reliably in ipython (GH22707) Bug in *read_json()* where default encoding was not set to utf-8 (GH29565)

Bug in *PythonParser* where str and bytes were being mixed when dealing with the decimal field (GH29650)

read_gbq() now accepts *progress_bar_type* to display progress bar while the

data downloads. (GH29857)

Bug in `.io.json.json_normalize()` where a missing value in the location specified by `record_path` would raise a `TypeError` (GH30148)

`read_excel()` now accepts binary data (GH15914)

Bug in `read_csv()` in which encoding handling was limited to just the string `utf-16` for the C engine

(GH24130)

5.3. Version 1.0

Plotting

Bug in `Series.plot()` not able to plot boolean values (GH23719)

Bug in `DataFrame.plot()` not able to plot when no rows (GH27758)

Bug in `DataFrame.plot()` producing incorrect legend markers when plotting multiple series on the same axis (GH18222)

Bug in `DataFrame.plot()` when `kind='box'` and data contains `datetime` or `timedelta` data. These types are now automatically dropped (GH22799)

Bug in `DataFrame.plot.line()` and `DataFrame.plot.area()` produce wrong `xlim` in x-axis (GH27686, GH25160, GH24784)

Bug where `DataFrame.boxplot()` would not accept a color parameter like `DataFrame.plot.box()` (GH26214)

Bug in the `xticks` argument being ignored for `DataFrame.plot.bar()` (GH14119)

`set_option()` now validates that the plot backend provided to `'plotting.backend'` implements the backend when the option is set, rather than when a plot is created (GH28163)

`DataFrame.plot()` now allow a `backend` keyword argument to allow changing between backends in one session (GH28619).

Bug in color validation incorrectly raising for non-color styles (GH29122).

Allow `DataFrame.plot.scatter()` to plot objects and `datetime` type data (GH18755, GH30391)

Bug in `DataFrame.hist()`, `xrot=0` does not work with `by` and subplots (GH30288). **GroupBy/resample/rolling**

Bug in `core.groupby.DataFrameGroupBy.apply()` only showing output from a

single group when function returns an *Index* (GH28652)

Bug in *DataFrame.groupby()* with multiple groups where an *IndexError* would be raised if any group contained all NA values (GH20519)

Bug in *.core.resample.Resampler.size()* and *.core.resample.*

Resampler.count() returning wrong dtype when used with an empty *Series* or *DataFrame* (GH28427)

Bug in *DataFrame.rolling()* not allowing for rolling over datetimes when *axis=1* (GH28192)

Bug in *DataFrame.rolling()* not allowing rolling over multi-index levels (GH15584). Bug in *DataFrame.rolling()* not allowing rolling on monotonic decreasing time indexes (GH19248).

639 Chapter 5. Release notes

Bug in *DataFrame.groupby()* not offering selection by column name when *axis=1* (GH27614)

Bug in *core.groupby.DataFrameGroupby.agg()* not able to use lambda function with named aggregation (GH27519)

Bug in *DataFrame.groupby()* losing column name information when grouping by a categorical column (GH28787)

Remove error raised due to duplicated input functions in named aggregation in *DataFrame.groupby()* and *Series.groupby()*. Previously error will be raised if the same function is applied on the same column and now it is allowed if new assigned names are different. (GH28426)

core.groupby.SeriesGroupBy.value_counts() will be able to handle the case even when the *Grouper* makes empty groups (GH28479)

Bug in *core.window.rolling.Rolling.quantile()* ignoring interpolation keyword argument when used within a *groupby* (GH28779)

Bug in *DataFrame.groupby()* where any, all, nunique and transform functions would incorrectly handle duplicate column labels (GH21668)

Bug in *core.groupby.DataFrameGroupBy.agg()* with timezone-aware *datetime64* column incorrectly casting results to the original dtype (GH29641)

Bug in *DataFrame.groupby()* when using *axis=1* and having a single level columns index (GH30208)

Bug in *DataFrame.groupby()* when using *nunique* on *axis=1* (GH30253) Bug in *GroupBy.quantile()* with multiple list-like *q* value and integer column names (GH30289)

Bug in *GroupBy.pct_change()* and *core.groupby.SeriesGroupBy.pct_change()* causes *TypeError* when *fill_method* is *None* (GH30463)

Bug in *Rolling.count()* and *Expanding.count()* argument where *min_periods* was ignored (GH26996)

Reshaping

Bug in *DataFrame.apply()* that caused incorrect output with empty *DataFrame* (GH28202, GH21959)

Bug in *DataFrame.stack()* not handling non-unique indexes correctly when creating *MultiIndex* (GH28301)

Bug in *pivot_table()* not returning correct type float when *margins=True* and *aggfunc='mean'* (GH24893)

Bug *merge_asof()* could not use *datetime.timedelta* for tolerance kwarg (GH28098) Bug in *merge()*, did not append suffixes correctly with *MultiIndex* (GH28518) *qcut()* and *cut()* now handle boolean input (GH20303)

Fix to ensure all int dtypes can be used in *merge_asof()* when using a tolerance value. Previously every non-int64 type would raise an erroneous *MergeError* (GH28870). Better error message in *get_dummies()* when columns isn't a list-like value (GH28383) Bug in *Index.join()* that caused infinite recursion error for mismatched *MultiIndex* name orders. (GH25760, GH28956)

Bug *Series.pct_change()* where supplying an anchored frequency would throw a *ValueError* (GH28664)

Bug where *DataFrame.equals()* returned *True* incorrectly in some cases when two *DataFrames* had the same columns in different orders (GH28839)

Bug in *DataFrame.replace()* that caused nonnumeric replacer's dtype not respected (GH26632)

Bug in *melt()* where supplying mixed strings and numeric values for *id_vars* or *value_vars* would incorrectly raise a *ValueError* (GH29718)

Dtypes are now preserved when transposing a *DataFrame* where each column

is the same extension dtype (GH30091)

Bug in *merge_asof()* merging on a tz-aware left_index and right_on a tz-aware column (GH29864)

5.3. Version 1.0

Improved error message and docstring in *cut()* and *qcut()* when labels=True (GH13318) Bug in missing fill_na parameter to *DataFrame.unstack()* with list of levels (GH30740) **Sparse**

Bug in SparseDataFrame arithmetic operations incorrectly casting inputs to float (GH28107)

Bug in DataFrame.sparse returning a Series when there was a column named sparse rather than the accessor (GH30758)

Fixed operator.xor() with a boolean-dtype SparseArray. Now returns a sparse result, rather than object dtype (GH31025)

ExtensionArray

Bug in *arrays.Array* when setting a scalar string (GH28118, GH28150). Bug where nullable integers could not be compared to strings (GH28930)

Bug where *DataFrame* constructor raised ValueError with list-like data and dtype specified (GH30280) **Other**

Trying to set the display.precision, display.max_rows or display.max_columns using *set_option()* to anything but a None or a positive int will raise a ValueError (GH23348) Using *DataFrame.replace()* with overlapping keys in a nested dictionary will no longer raise, now matching the behavior of a flat dictionary (GH27660)

DataFrame.to_csv() and *Series.to_csv()* now support dicts as compression argument with key 'method' being the compression method and others as additional compression options when the compression method is 'zip'. (GH26023)

Bug in *Series.diff()* where a boolean series would incorrectly raise a TypeError (GH17294)

Series.append() will no longer raise a TypeError when passed a tuple of Series (GH28410)

Fix corrupted error message when calling *.libs._json.encode()* on a 0d array

(GH18878)

Backtick quoting in *DataFrame.query()* and *DataFrame.eval()* can now also be used to use invalid identifiers like names that start with a digit, are python keywords, or are using single character operators. (GH27017)

Bug in *pd.core.util.hashing.hash__* object where arrays containing tuples were incorrectly treated as non-hashable (GH28969)

Bug in *DataFrame.append()* that raised *IndexError* when appending with empty list (GH28769)

Fix *AbstractHolidayCalendar* to return correct results for years after 2030 (now goes up to 2200) (GH27790)

Fixed *IntegerArray* returning *inf* rather than *NaN* for operations dividing by 0 (GH27398) Fixed *pow* operations for *IntegerArray* when the other value is 0 or 1 (GH29997) Bug in *Series.count()* raises if *use_inf_as_na* is enabled (GH29478)

Bug in *Index* where a non-hashable name could be set without raising *TypeError* (GH29069)

Bug in *DataFrame* constructor when passing a 2D ndarray and an extension dtype (GH12513)

Bug in *DataFrame.to_csv()* when supplied a series with a *dtype="string"* and a *na_rep*, the *na_rep* was being truncated to 2 characters. (GH29975)

Bug where *DataFrame.itertuples()* would incorrectly determine whether or not *namedtuples* could be used for dataframes of 255 columns (GH28282)

Handle nested NumPy object arrays in *testing.assert_series_equal()* for *ExtensionArray* implementations (GH30841)

Bug in *Index* constructor incorrectly allowing 2-dimensional input arrays (GH13601, GH27125)

Contributors

A total of 308 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Aaditya Panikath +
Abdullah İhsan Seçer
Abhijeet Krishnan +

Adam J. Stewart
Adam Klaum +
Addison Lynch
Aivengoe +
Alastair James +
Albert Villanova del Moral
Alex Kirko +
Alfredo Granja +
Allen Downey
Alp Arıbal +
Andreas Buhr +
Andrew Munch +
Andy
Angela Ambroz +
Aniruddha Bhattacharjee +

-
-

Ankit Dhankhar +

Antonio Andraues Jr + Arda Kosar +
Asish Mahapatra + Austin Hackett +
Avi Kelman +
AyowoleT +

5.3. Version 1.0

Bas Nijholt +
Ben Thayer
Bharat Raghunathan Bhavani Ravi
Bhuvana KA +
Big Head
Blake Hawkins +
Bobae Kim +
Brett Naul
Brian Wignall
Bruno P. Kinoshita + Bryant Moscon +
Cesar H +
Chris Stadler

Chris Zimmerman +

-
-
-
-

-

Christopher Whelan

Clemens Brunner

Clemens Tolboom +

Connor Charles +

Daniel Hähnke +

Daniel Saxton

Darin Plutchok +

Dave Hughes

David Stansby

DavidRosen + • Dean +

Deepan Das +

Deepyaman Datta

DorAmram +

Dorothy Kabarozi +

Drew Heenan +

Eliza Mae Saret +

Elle +

Endre Mark Borza + Eric Brassell + Eric Wong +

Eunseop Jeong +

Eyden Villanueva +

Felix Divo

ForTimeBeing +

-
-
-
-
-

Francesco Truzzi +

Gabriel Corona +

Gabriel Monteiro +

Galuh Sahid +

Georgi Baychev +

Gina

GiuPassarelli +

Grigorios Giannakopoulos + Guilherme Leite +

Guilherme Salomé +

Gyeongjae Choi +

Harshavardhan Bachina + Harutaka Kawamura + Hassan Kibirige

Hielke Walinga

Hubert

Hugh Kelley +

Ian Eaves +

Ignacio Santolin +

Igor Filippov +

Irv Lustig

Isaac Virshup +

Ivan Bessarabov +

-
-
-
-

-

JMBurley +

Jack Bicknell +

Jacob Buckheit +

Jan Koch

Jan Pipek + Jan Škoda + Jan-Philip Gehrcke

Jasper J.F. van den Bosch + Javad +

Jeff Reback

Jeremy Schendel

Jeroen Kant +

Jesse Pardue +
Jethro Cao +
Jiang Yue
Jiaxiang +
Jihyung Moon +
Jimmy Callin
Jinyang Zhou +
Joao Victor Martinelli +
Joaq Almirante +
John G Evans +
John Ward +
Jonathan Larkin +

•
•
•
•

•

Joris Van den Bossche

Josh Dimarsky +
Joshua Smith +
Josiah Baker +
Julia Signell +
Jung Dong Ho +
Justin Cole +
Justin Zheng
Kaiqi Dong
Karthigeyan +
Katherine Younglove +
Katrin Leinweber
Kee Chong Tan +
Keith Kraus +
Kevin Nguyen +
Kevin Sheppard
Kisekka David +

Koushik +
Kyle Boone +
Kyle McCahill +
Laura Collard, PhD + LiuSeeker + Louis Huynh +
Lucas Scarlato Astur +

-
-
-
-

•
Luiz Gustavo +

Luke +
Luke Shepard +
MKhalusova +
Mabel Villalba • Maciej J + Mak Sze Chun
Manu NALEPA +
Marc
Marc Garcia
Marco Gorelli +
Marco Neumann +
Martin Winkel +
Martina G. Vilas +
Mateusz +
Matthew Roeschke
Matthew Tan +
Max Bolingbroke
Max Chen +
MeeseeksMachine
Miguel +
MinGyo Jung +
Mohamed Amine ZGHAL + Mohit Anand +

-
-
-

-

-

MomIsBestFriend +

Naomi Bonnin +

Nathan Abel +

Nico Cernek +

Nigel Markey +

Noritada Kobayashi +

Oktay Sabak +

Oliver Hofkens +

Oluokun Adedayo +

Osman +

Oguzhan Özgreden + Development Team +

Patrik Hlobil +

Paul Lee +

Paul Siegel +

Petr Baev +

Pietro Battiston

Prakhar Pandey +

Puneeth K +

Raghav +

Rajat +

Rajhans Jadhao +

Rajiv Bharadwaj +

-

-

-

-

-

Rik-de-Kort +

Roei.r

Rohit Sanjay +

Ronan Lamy +
Roshni +
Roymprog +
Rushabh Vasani +
Ryan Grout +
Ryan Nazareth
Samesh Lakhotia +
Samuel Sinayoko
Samyak Jain +
Sarah Donehower +
Sarah Masud +
Saul Shanabrook +
Scott Cole +
SdgJlbl +
Seb +
Sergei Ivko +
Shadi Akiki
Shorokhov Sergey
Siddhesh Poyarekar + Sidharthan Nair + Simon Gibbons

-
-
-
-

-

Simon Hawkins

Simon-Martin Schröder + Sofiane Mahiou +
Sourav kumar +
Souvik Mandal +
Soyoun Kim +
Sparkle Russell-Puleri + Srinivas Reddy Thatiparthi () Stuart Berg +
Sumanau Sareen
Szymon Bednarek +
Tambe Tabitha Achere + Tan Tran
Tang Heyi +

Tanmay Daripa +
Tanya Jain
Terji Petersen
Thomas Li +
Tirth Jain +
Tola A +
Tom Augspurger
Tommy Lynch +
Tomoyuki Suzuki +

•
•
•
•

•

Tony Lorenzo

Unprocessable +
Uwe L. Korn
Vaibhav Vishal
Victoria Zdanovskaya + Vijayant +
Vishwak Srinivasan + WANG Aiyong Wenhuan Wes McKinney
Will Ayd
Will Holmgren
William Ayd
William Blan +
Wouter Overmeire
Wuraola Oyewusi +
YaOzI +
Yash Shukla +
Yu Wang +
Yusei Tahara +
alexander135 +
alimcmaster1
avelineg +
bganglia +

-
-
-
-

-

bolkedebruin

bravech +

chinhwee +

cruzzoe +

dalgarno +

daniellebrown +

danielplawrence

est271 +

francisco souza +

ganevgv +

garanews +

gfyong

h-vetinari

hasnain2808 +

ianzur +

jalbritt +

jbrockmendel

jeschwar +

jlamborn324 + joy-rosie + kernc killerontherun1 krey +

-
-
-
-

-

lexy-lixinyu + lucyleeow + lukasbk + maheshbapatu +

mck619 +

nathalier

naveenkaushik2504 +

nlepleux +
nrebena
ohad83 +
pilkibun
pqzx +
proost +
pv8493013j +
qudade +
rhstanton +
rmunjal29 +
sangarshanan +
sardonick +
saskakarsi +
shaido987 +
ssikdar1
steveayers124 +
tadashigaki +
timcera +
tlaytongoogle +
tobycheese
tonywu1999 +
tsvikas +
yogendrasoni +

5.3. Version 1.0 655

zys5945 +

5.4 Version 0.25

5.4.1 What's new in 0.25.3 (October 31, 2019)

These are the changes in 0.25.3. See *Release notes* for a full changelog including other versions of .

Bug fixes

GroupBy/resample/rolling

- Bug in DataFrameGroupBy.quantile() where NA values in the grouping could cause segfaults or incorrect results (GH28882)

Contributors

A total of 2 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Will Ayd

William Ayd

5.4.2 What’s new in 0.25.2 (October 15, 2019)

These are the changes in 0.25.2. See *Release notes* for a full changelog including other versions of .

Note: 0.25.2 adds compatibility for Python 3.8 (GH28147).

Bug fixes Indexing

Fix regression in `DataFrame.reindex()` not following the limit argument (GH28631).

Fix regression in `RangeIndex.get_indexer()` for decreasing `RangeIndex` where target values may be improperly identified as missing/present (GH28678)

IO

Fix regression in notebook display where `<th>` tags were missing for `DataFrame.index` values (GH28204).

Regression in `to_csv()` where writing a `Series` or `DataFrame` indexed by an `IntervalIndex` would incorrectly raise a `TypeError` (GH28210)

Fix `to_csv()` with `ExtensionArray` with list-like values (GH28840).

GroupBy/resample/rolling

Bug incorrectly raising an `IndexError` when passing a list of quantiles to `.core.groupby.DataFrameGroupBy.quantile()` (GH28113).

Bug in `.core.groupby.GroupBy.shift()`, `.core.groupby.GroupBy.bfill()` and `.core.groupby.GroupBy.ffill()` where timezone information would be dropped (GH19995, GH27992)

Other

Compatibility with Python 3.8 in `DataFrame.query()` (GH27261)

Fix to ensure that tab-completion in an IPython console does not raise warnings for deprecated attributes (GH27900).

Contributors

A total of 6 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Felix Divo +

Jeremy Schendel

Joris Van den Bossche

MeeseeksMachine

Tom Augspurger

jbrockmendel

5.4.3 What’s new in 0.25.1 (August 21, 2019)

These are the changes in 0.25.1. See *Release notes* for a full changelog including other versions of .

IO and LZMA

Some users may unknowingly have an incomplete Python installation lacking the lzma module from the standard library. In this case, import failed due to an ImportError (GH27575). will now warn, rather than raising an ImportError if the lzma module is not present. Any subsequent attempt to use lzma methods will raise a RuntimeError. A possible fix for the lack of the lzma module is to ensure you have the necessary libraries and then re-install Python. For example, on MacOS installing Python with pyenv may lead to an incomplete Python installation due to unmet system dependencies at compilation time (like xz). Compilation will succeed, but Python might fail at run time. The issue can be solved by installing the necessary dependencies and then re-installing Python.

5.4. Version 0.25

Bug fixes

Categorical

Bug in Categorical.fillna() that would replace all values, not just those that are NaN (GH26215)

Datetimelike

Bug in to_datetime() where passing a timezone-naive DatetimeArray or DatetimeIndex and utc=True would incorrectly return a timezone-naive result (GH27733)

Bug in `Period.to_timestamp()` where a `Period` outside the `Timestamp` implementation bounds (roughly 1677-09-21 to 2262-04-11) would return an incorrect `Timestamp` instead of raising `OutOfBoundsDatetime` (GH19643)

Bug in iterating over `DatetimeIndex` when the underlying data is read-only (GH28055) **Timezones**

Bug in `Index` where a numpy object array with a timezone aware `Timestamp` and `np.nan` would not return a `DatetimeIndex` (GH27011)

Numeric

Bug in `Series.interpolate()` when using a timezone aware `DatetimeIndex` (GH27548)

Bug when printing negative floating point complex numbers would raise an `IndexError` (GH27484)

Bug where `DataFrame` arithmetic operators such as `DataFrame.mul()` with a `Series` with `axis=1` would raise an `AttributeError` on `DataFrame` larger than the minimum threshold to invoke `numexpr` (GH27636)

Bug in `DataFrame` arithmetic where missing values in results were incorrectly masked with `NaN` instead of

`Inf` (GH27464)

Conversion

Improved the warnings for the deprecated methods `Series.real()` and `Series.imag()` (GH27610)

Interval

Bug in `IntervalIndex` where `dir(obj)` would raise `ValueError` (GH27571)

Indexing

Bug in partial-string indexing returning a NumPy array rather than a `Series` when indexing with a scalar like `.loc['2015']` (GH27516)

Break reference cycle involving `Index` and other index classes to allow garbage collection of index objects without running the GC. (GH27585,

GH27840) Fix regression in assigning values to a single column of a DataFrame with a MultiIndex columns (GH27841).

Fix regression in .ix fallback with an IntervalIndex (GH27865).

Missing

Bug in *.isnull()* or *.isna()* when the input is a type e.g. type(.Series()) (GH27482)

IO

Avoid calling S3File.s3 when reading parquet, as this was removed in s3fs version 0.3.0 (GH27756)

Better error message when a negative header is passed in *.read_csv()* (GH27779) Follow the min_rows display option (introduced in v0.25.0) correctly in the HTML repr in the notebook (GH27991).

Plotting

Added a *_plotting_backends* entrypoint group for registering plot backends. See *Plotting backends* for more (GH26747).

Fixed the re-instatement of Matplotlib datetime converters after calling *.plotting.deregister_matplotlib_converters()* (GH27481).

Fix compatibility issue with matplotlib when passing a Index to a plot call (GH27775). **GroupBy/resample/rolling**

Fixed regression in *pandas.core.groupby.DataFrameGroupBy.quantile()* raising when multiple quantiles are given (GH27526)

Bug in *.core.groupby.DataFrameGroupBy.transform()* where applying a timezone conversion lambda function would drop timezone information (GH27496)

Bug in *.core.groupby.GroupBy.nth()* where *observed=False* was being ignored for Categorical groupers (GH26385)

Bug in windowing over read-only arrays (GH27766)

Fixed segfault in *.core.groupby.DataFrameGroupBy.quantile* when an invalid quantile was passed (GH27470)

5.4. Version 0.25

Reshaping

A `KeyError` is now raised if `.unstack()` is called on a `Series` or `DataFrame` with a flat `Index` passing a name which is not the correct one (GH18303)
Bug `merge_asof()` could not merge `Timedelta` objects when passing tolerance kwarg (GH27642)

Bug in `DataFrame.crosstab()` when margins set to `True` and `normalize` is not `False`, an error is raised. (GH27500)

`DataFrame.join()` now suppresses the `FutureWarning` when the `sort` parameter is specified (GH21952)

Bug in `DataFrame.join()` raising with readonly arrays (GH27943)

Sparse

Bug in reductions for `Series` with `Sparse` dtypes (GH27080)

Other

Bug in `Series.replace()` and `DataFrame.replace()` when replacing timezone-aware timestamps using a dict-like replacer (GH27720)

Bug in `Series.rename()` when using a custom type indexer. Now any value that isn't callable or dict-like is treated as a scalar. (GH27814)

Contributors

A total of 5 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Jeff Reback

Joris Van den Bossche

MeeseeksMachine +

Tom Augspurger

jbrockmendel

5.4.4 What's new in 0.25.0 (July 18, 2019)

Warning: Starting with the 0.25.x series of releases, only supports Python 3.5.3 and higher. See [Dropping Python 2.7](#) for more details.

Warning: The minimum supported Python version will be bumped to 3.6 in a future release.

Warning: `Panel` has been fully removed. For N-D labeled data structures, please use `xarray`

Warning: `read_pickle()` and `read_msgpack()` are only guaranteed backwards compatible back to version 0.20.3 (GH27082)

These are the changes in 0.25.0. See *Release notes* for a full changelog including other versions of .

Enhancements

GroupBy aggregation with relabeling

has added special groupby behavior, known as “named aggregation”, for naming the output columns when applying multiple aggregation functions to specific columns (GH18366, GH26512).

```
In [1]: animals = pd.DataFrame({'kind': ['cat', 'dog', 'cat', 'dog'],  
...: 'height': [9.1, 6.0, 9.5, 34.0], ...: 'weight': [7.9, 7.5, 9.9, 198.0]}) ...:
```

```
In [2]: animals Out[2]:
```

```
kind height weight
```

```
cat 9.1 7.9
```

```
dog 6.0 7.5
```

```
cat 9.5 9.9
```

```
dog 34.0 198.0
```

```
[4 rows x 3 columns]
```

```
In [3]: animals.groupby("kind").agg(
```

```
...: min_height=pd.NamedAgg(column='height', aggfunc='min'), ...:
```

```
max_height=pd.NamedAgg(column='height', aggfunc='max'), ...:
```

```
average_weight=pd.NamedAgg(column='weight', aggfunc=np.mean), ...: )
```

```
...: Out[3]: min_height max_height average_weight
```

```
kind
```

```
cat 9.1 9.5 8.90 dog 6.0 34.0 102.75
```

```
[2 rows x 3 columns]
```

Pass the desired columns names as the ****kwargs** to `.agg`. The values of ****kwargs** should be tuples where the first element is the column selection, and the second element is the aggregation function to apply. `.NamedAgg` namedtuple to make it clearer what the arguments to the function are, but plain tuples are accepted as well.

```
In [4]: animals.groupby("kind").agg(
```

```
...: min_height=('height', 'min'),
```

```
...: max_height=('height', 'max'),
```

```
...: average_weight=('weight', np.mean),
```

```
...: ) ...:
```

Out[4]:

```
min_height max_height average_weight
```

(continues on next page)

5.4. Version 0.25

(continued from previous page)

```
kind cat 9.1 9.5 8.90
```

```
dog 6.0 34.0 102.75
```

```
[2 rows x 3 columns]
```

Named aggregation is the recommended replacement for the deprecated “dict-of-dicts” approach to naming the output of column-specific aggregations (*Deprecate `groupby.agg()` with a dictionary when renaming*).

A similar approach is now available for Series groupby objects as well. Because there’s no need for column selection, the values can just be the functions to apply

In [5]: `animals.groupby("kind").height.agg(`

```
...: min_height="min",
```

```
...: max_height="max",
```

```
...: ) ...:
```

Out[5]: `min_height max_height`

```
kind
```

```
cat 9.1 9.5 dog 6.0 34.0
```

```
[2 rows x 2 columns]
```

This type of aggregation is the recommended alternative to the deprecated behavior when passing a dict to a Series groupby aggregation (*Deprecate `groupby.agg()` with a dictionary when renaming*).

See *Named aggregation* for more.

GroupBy aggregation with multiple lambdas

You can now provide multiple lambda functions to a list-like aggregation in `.core.groupby.GroupBy.agg` (GH26430).

```
In [6]: animals.groupby('kind').height.agg([
...: lambda x: x.iloc[0], lambda x: x.iloc[-1]
...: ]) ...:
```

```
Out[6]:
```

```
<lambda_0> <lambda_1> kind
cat 9.1 9.5 dog 6.0 34.0
[2 rows x 2 columns]
```

```
In [7]: animals.groupby('kind').agg([
...: lambda x: x.iloc[0] - x.iloc[1],
...: lambda x: x.iloc[0] + x.iloc[1]
...: ]) ...:
```

```
Out[7]:
```

```
height weight
<lambda_0> <lambda_1> <lambda_0> <lambda_1>
(continues on next page)
(continued from previous page)
```

```
kind cat -0.4 18.6 -2.0 17.8
dog -28.0 40.0 -190.5 205.5
```

```
[2 rows x 4 columns]
```

Previously, these raised a `SpecificationError`.

Better repr for MultiIndex

Printing of *MultiIndex* instances now shows tuples of each row and ensures that the tuple items are vertically aligned, so it's now easier to understand the structure of the *MultiIndex*. (GH13480):

The repr now looks like this:

```
In [8]: pd.MultiIndex.from_product(['a', 'abc'], range(500)) Out[8]:
MultiIndex([( 'a', 0),
```

```
( 'a', 1),
( 'a', 2),
( 'a', 3),
( 'a', 4),
( 'a', 5),
( 'a', 6),
```

```
( 'a', 7),
( 'a', 8),
( 'a', 9),

...
('abc', 490),
('abc', 491),
('abc', 492),
('abc', 493),
('abc', 494),
('abc', 495),
('abc', 496),
('abc', 497),
('abc', 498),
('abc', 499)], length=1000)
```

Previously, outputting a *MultiIndex* printed all the levels and codes of the MultiIndex, which was visually unappealing and made the output more difficult to navigate. For example (limiting the range to 5):

```
In [1]: pd.MultiIndex.from_product(['a', 'abc'], range(5)) Out[1]:
MultiIndex(levels= [['a', 'abc'], [0, 1, 2, 3]],
...: codes= [[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])
```

In the new repr, all values will be shown, if the number of rows is smaller than `options.display.max_seq_items` (default: 100 items). Horizontally, the output will truncate, if it's wider than `options.display.width` (default: 80 characters).

5.4. Version 0.25

Shorter truncated repr for Series and DataFrame

Currently, the default display options of ensure that when a Series or DataFrame has more than 60 rows, its repr gets truncated to this maximum of 60 rows (the `display.max_rows` option). However, this still gives a repr that takes up a large part of the vertical screen estate. Therefore, a new option `display.min_rows` is introduced with a default of 10 which determines the number of rows showed in the truncated repr:

For small Series or DataFrames, up to `max_rows` number of rows is shown (default: 60).

For larger Series of DataFrame with a length above `max_rows`, only `min_rows` number of rows is shown

(default: 10, i.e. the first and last 5 rows).

This dual option allows to still see the full content of relatively small objects (e.g. `df.head(20)` shows all 20 rows), while giving a brief repr for large objects. To restore the previous behaviour of a single threshold, set `pd.options.display.min_rows = None`.

JSON normalize with max_level param support

`json_normalize()` normalizes the provided input dict to all nested levels. The new `max_level` parameter provides more control over which level to end normalization (GH23843):

The repr now looks like this:

```
from .io.json import json_normalize data = [{
'CreatedBy': {'Name': 'User001'},
'Lookup': {'TextField': 'Some text',
'UserField': {'Id': 'ID001', 'Name': 'Name001'}},
'Image': {'a': 'b'}
}] json_normalize(data, max_level=1)
```

Series.explode to split list-like values to rows

Series and *DataFrame* have gained the *DataFrame.explode()* methods to transform listlikes to individual rows. See *section on Exploding list-like column* in docs for more information (GH16538, GH10511) Here is a typical usecase. You have comma separated string in a column.

```
In [9]: df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1},
```

```
...: {'var1': 'd,e,f', 'var2': 2}])
```

```
...:
```

```
In [10]: df Out[10]: var1 var2
```

```
a,b,c 1
```

```
d,e,f 2
```

```
[2 rows x 2 columns]
```

Creating a long form DataFrame is now straightforward using chained operations **In [11]:** `df.assign(var1=df.var1.str.split(',')).explode('var1')`

Out[11]:

var1 var2

0 a 1

0 b 1

c 1

d 2

1 e 2

1 f 2

[6 rows x 2 columns]

Other enhancements

DataFrame.plot() keywords *logy*, *logx* and *loglog* can now accept the value 'sym' for symlog scaling. (GH24867)

Added support for ISO week year format ('%G-%V%u') when parsing datetimes using *to_datetime()* (GH16607)

Indexing of *DataFrame* and *Series* now accepts zerodim *np.ndarray* (GH24919) *Timestamp.replace()* now supports the *fold* argument to disambiguate DST transition times (GH25017)

DataFrame.at_time() and *Series.at_time()* now support *datetime.time* objects with timezones (GH24043)

DataFrame.pivot_table() now accepts an *observed* parameter which is passed to underlying calls to *DataFrame.groupby()* to speed up grouping categorical data. (GH24923)

Series.str has gained *Series.str.casefold()* method to removes all case distinctions present in a string (GH25405)

DataFrame.set_index() now works for instances of *abc.Iterator*, provided their output is of the same length as the calling frame (GH22484, GH24984)

DatetimeIndex.union() now supports the *sort* argument. The behavior of the *sort* parameter matches that of *Index.union()* (GH24994)

RangeIndex.union() now supports the *sort* argument. If *sort=False* an unsorted *Int64Index* is always returned. *sort=None* is the default and returns a monotonically increasing *RangeIndex* if possible or a sorted *Int64Index* if not (GH24471)

TimedeltaIndex.intersection() now also supports the *sort* keyword

(GH24471) *DataFrame.rename()* now supports the *errors* argument to raise errors when attempting to rename nonexistent keys (GH13473)

Added *Sparse accessor* for working with a *DataFrame* whose values are sparse (GH25681)

RangeIndex has gained *start*, *stop*, and *step* attributes (GH25710)

datetime.timezone objects are now supported as arguments to *timezone* methods and constructors (GH25065)

DataFrame.query() and *DataFrame.eval()* now supports quoting column names with backticks to refer to names with spaces (GH6508)

merge_asof() now gives a more clear error message when merge keys are categoricals that are not equal

(GH26136)

5.4. Version 0.25

.core.window.Rolling() supports exponential (or Poisson) window type (GH21303)

Error message for missing required imports now includes the original import error's text (GH23868)

DatetimeIndex and *TimedeltaIndex* now have a *mean* method (GH24757)

DataFrame.describe() now formats integer percentiles without decimal point (GH26660) Added support for reading SPSS .sav files using *read_spss()* (GH26537)

Added new option *plotting.backend* to be able to select a plotting backend different than the existing matplotlib one. Use *.set_option('plotting.backend', '<backend-module>')* where *<backend-module>* is a library implementing the plotting API (GH14130)

.offsets.BusinessHour supports multiple opening hours intervals (GH15481)

read_excel() can now use *openpyxl* to read Excel files via the *engine='openpyxl'* argument. This will become the default in a future release (GH11499)

.io.excel.read_excel() supports reading OpenDocument tables. Specify *engine='odf'* to enable. Consult the *IO User Guide* for more details (GH9070)

Interval, *IntervalIndex*, and *IntervalArray* have gained an *is_empty* attribute

denoting if the given interval(s) are empty (GH27219)

Backwards incompatible API changes

Indexing with date strings with UTC offsets

Indexing a *DataFrame* or *Series* with a *DatetimeIndex* with a date string with a UTC offset would previously ignore the UTC offset. Now, the UTC offset is respected in indexing. (GH24076, GH16785)

```
In [12]: df = pd.DataFrame([0], index=pd.DatetimeIndex(['2019-01-01'],
tz='US/Pacific', →'))
```

```
In [13]: df Out[13]:
```

```
0
2019-01-01 00:00:00-08:00 0
[1 rows x 1 columns]
```

Previous behavior:

```
In [3]: df['2019-01-01 00:00:00+04:00':'2019-01-01 01:00:00+04:00']
```

```
Out[3]:
```

```
0
2019-01-01 00:00:00-08:00 0
```

New behavior:

```
In [14]: df['2019-01-01 12:00:00+04:00':'2019-01-01 13:00:00+04:00']
```

```
Out[14]:
```

```
0
2019-01-01 00:00:00-08:00 0
[1 rows x 1 columns]
```

MultiIndex constructed from levels and codes

Constructing a *MultiIndex* with NaN levels or codes value < -1 was allowed previously. Now, construction with codes value < 1 is not allowed and NaN levels' corresponding codes would be reassigned as -1. (GH19387) *Previous behavior:*

```
In [1]: pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
```

```
...: codes=[[0, -1, 1, 2, 3, 4]])
```

```
...:
```

```
Out[1]: MultiIndex(levels=[[nan, None, NaT, 128, 2]], codes=[[0, -1, 1, 2, 3,
```



```
4]])
```

```
In [2]: pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]]) Out[2]:  
MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
```

New behavior:

```
In [15]: pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
```

```
....: codes=[[0, -1, 1, 2, 3, 4]])
```

```
....:
```

```
Out[15]:
```

```
MultiIndex([(nan,),  
(nan,),  
(nan,),  
(128,),  
( 2,)],  
)
```

```
In [16]: pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
```

```
ValueError Traceback (most recent call last)
```

```
<ipython-input-16-225a01af3975> in <module>
```

```
----> 1 pd.MultiIndex(levels=[[1, 2]], codes=[[0, -2]])
```

```
///core/indexes/multi.py in __new__(cls, levels, codes, sortorder, names,  
, dtype, copy, name, verify_integrity) 314
```

```
315 if verify_integrity:
```

```
--> 316 new_codes = result._verify_integrity()
```

```
317 result._codes = new_codes
```

```
318
```

```
///core/indexes/multi.py in _verify_integrity(self, codes, levels) )
```

```
if len(level_codes) and level_codes.min() < -1:
```

```
--> 389 raise ValueError(f"On level {i}, code value ({level_codes. , - min()})  
< -1")
```

```
if not level.is_unique:
```

```
raise ValueError(
```

```
ValueError: On level 0, code value (-2) < -1
```

5.4. Version 0.25

GroupBy.apply on DataFrame evaluates first group only once

The implementation of `DataFrameGroupBy.apply()` previously evaluated the supplied function consistently twice on the first group to infer if it is safe to use a fast code path. Particularly for functions with side effects, this was an undesired behavior and may have led to surprises. (GH2936, GH2656, GH7739, GH10519, GH12155, GH20084, GH21417)

Now every group is evaluated only a single time.

In [17]: `df = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})`

In [18]: `df` **Out[18]:** a b 0 x 1

1 y 2

[2 rows x 2 columns]

In [19]: `def func(group):`

`....: print(group.name)`

`....: return group`

`....:`

Previous behavior:

In [3]: `df.groupby('a').apply(func)` x x y **Out[3]:** a b 0 x 1

1 y 2

New behavior:

In [20]: `df.groupby("a").apply(func)` x y **Out[20]:** a b 0 x 1

1 y 2

[2 rows x 2 columns]

Concatenating sparse values

When passed DataFrames whose values are sparse, `concat()` will now return a *Series* or *DataFrame* with sparse values, rather than a `SparseDataFrame` (GH25702).

In [21]: `df = pd.DataFrame({"A": pd.SparseArray([0, 1])})`

Previous behavior:

In [2]: `type(pd.concat([df, df]))` `.core.sparse.frame.SparseDataFrame`

New behavior:

In [22]: `type(pd.concat([df, df]))`

Out[22]: `.core.frame.DataFrame`

This now matches the existing behavior of *concat* on Series with sparse values. *concat()* will continue to return a SparseDataFrame when all the values are instances of SparseDataFrame.

This change also affects routines using *concat()* internally, like *get_dummies()*, which now returns a *DataFrame* in all cases (previously a SparseDataFrame was returned if all the columns were dummy encoded, and a *DataFrame* otherwise).

Providing any SparseSeries or SparseDataFrame to *concat()* will cause a SparseSeries or SparseDataFrame to be returned, as before.

The .str-accessor performs stricter type checks

Due to the lack of more fine-grained dtypes, *Series.str* so far only checked whether the data was of object dtype. *Series.str* will now infer the dtype data *within* the Series; in particular, 'bytes'-only data will raise an exception (except for *Series.str.decode()*, *Series.str.get()*, *Series.str.len()*, *Series.str.slice()*), see GH23163, GH23011, GH23551.

Previous behavior :

```
In [1]: s = pd.Series(np.array(['a', 'ba', 'cba'], 'S'), dtype=object)
```

```
In [2]: s
```

```
Out[2]:
```

```
b'a'
```

```
b'ba' 2 b'cba'
```

```
dtype: object
```

```
In [3]: s.str.startswith(b'a') Out[3]:
```

```
True
```

```
False
```

```
False
```

```
dtype: bool
```

New behavior:

```
In [23]: s = pd.Series(np.array(['a', 'ba', 'cba'], 'S'), dtype=object)
```

```
(continues on next page)
```

5.4. Version 0.25

(continued from previous page)

```
In [24]: s
```

Out[24]:

b'a'

b'ba'

b'cba'

Length: 3, dtype: object

In [25]: s.str.startswith(b'a')

TypeError Traceback (most recent call last)

<ipython-input-25-ac784692b361> **in** <module>

----> 1 s.str.startswith(b'a')

///core/strings/accessor.py in wrapper(self, *args, **kwargs)

f"inferred dtype '{self._inferred_dtype}'."

)

--> 100 **raise** TypeError(msg)

101 **return** func(self, *args, **kwargs)

102

TypeError: Cannot use .str.startswith with values of inferred dtype 'bytes'.

Categorical dtypes are preserved during GroupBy

Previously, columns that were categorical, but not the groupby key(s) would be converted to object dtype during groupby operations. now will preserve these dtypes. (GH18502)

In [26]: cat = pd.Categorical(["foo", "bar", "bar", "qux"], ordered=**True**)

In [27]: df = pd.DataFrame({'payload': [-1, -2, -1, -2], 'col': cat})

In [28]: df **Out[28]:**

payload col

-1 foo

-2 bar

-1 bar

-2 qux

[4 rows x 2 columns]

In [29]: df.dtypes **Out[29]:**

payload int64 col category

Length: 2, dtype: object

Previous Behavior:

```
In [5]: df.groupby('payload').first().col.dtype
```

```
Out[5]: dtype('O')
```

New Behavior:

```
In [30]: df.groupby('payload').first().col.dtype
```

```
Out[30]: CategoricalDtype(categories=['bar', 'foo', 'qux'], ordered=True)
```

Incompatible Index type unions

When performing *Index.union()* operations between objects of incompatible dtypes, the result will be a base *Index* of dtype object. This behavior holds true for unions between *Index* objects that previously would have been prohibited. The dtype of empty *Index* objects will now be evaluated before performing union operations rather than simply returning the other *Index* object. *Index.union()* can now be considered commutative, such that *A.union(B) == B.union(A)* (GH23525).

Previous behavior :

```
In [1]: pd.period_range('19910905', periods=2).union(pd.Int64Index([1, 2, 3])) ... ValueError: can only call with other PeriodIndex-ed objects
```

```
In [2]: pd.Index([], dtype=object).union(pd.Index([1, 2, 3]))
```

```
Out[2]: Int64Index([1, 2, 3], dtype='int64')
```

New behavior:

```
In [31]: pd.period_range('19910905', periods=2).union(pd.Int64Index([1, 2, 3])) Out[31]: Index([1991-09-05, 1991-09-06, 1, 2, 3], dtype='object')
```

```
In [32]: pd.Index([], dtype=object).union(pd.Index([1, 2, 3]))
```

```
Out[32]: Index([1, 2, 3], dtype='object')
```

Note that integer- and floatingdtype indexes are considered “compatible”. The integer values are coerced to floating point, which may result in loss of precision. See *Set operations on Index objects* for more.

DataFrame GroupBy ffill/bfill no longer return group labels

The methods ffill, bfill, pad and backfill of DataFrameGroupBy previously included the group labels in the return value, which was inconsistent with other groupby transforms. Now only the filled values are returned. (GH21521)

In [33]: df = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})

In [34]: df **Out[34]:** a b 0 x 1

1 y 2

[2 rows x 2 columns]

Previous behavior :

In [3]: df.groupby("a").ffill() **Out[3]:** a b 0 x 1

1 y 2

5.4. Version 0.25

New behavior:

In [35]: df.groupby("a").ffill() **Out[35]:** b

1

2

[2 rows x 1 columns]

DataFrame describe on an empty Categorical / object column will return top and freq

When calling *DataFrame.describe()* with an empty categorical / object column, the 'top' and 'freq' columns were previously omitted, which was inconsistent with the output for nonempty columns. Now the 'top' and 'freq' columns will always be included, with `numpy.nan` in the case of an empty *DataFrame* (GH26397)

In [36]: df = pd.DataFrame({"empty_col": pd.Categorical([])})

In [37]: df **Out[37]:**

Empty DataFrame

Columns: [empty_col]

Index: []

[0 rows x 1 columns]

Previous behavior:

In [3]: df.describe()

Out[3]:

empty_col count

0 unique

0

New behavior:

In [38]: df.describe()

Out[38]:

```
empty_col count 0
unique 0 top
NaN freq
NaN
```

[4 rows x 1 columns]

__str__ methods now call __repr__ rather than vice versa

has until now mostly defined string representations in a objects'

__str__/__unicode__/__bytes__ methods, and called __str__ from the __repr__ method, if a specific __repr__ method is not found. This is not needed for Python3. In 0.25, the string representations of objects are now generally defined in __repr__, and calls to __str__ in general now pass the call on to the __repr__, if a specific __str__ method doesn't exist, as is standard for Python. This change is backward compatible for direct usage of , but if you subclass objects *and* give your subclasses specific __str__/__repr__ methods, you may have to adjust your __str__/__repr__ methods (GH26495).

Indexing an IntervalIndex with Interval objects

Indexing methods for *IntervalIndex* have been modified to require exact matches only for *Interval* queries. *IntervalIndex* methods previously matched on any overlapping *Interval*. Behavior with scalar points, e.g. querying with an integer, is unchanged (GH16316).

In [39]: ii = pd.IntervalIndex.from_tuples([(0, 4), (1, 5), (5, 8)]) **In [40]:** ii

Out[40]:

```
IntervalIndex([(0, 4], (1, 5], (5, 8]], closed='right', dtype='interval[int64]')
```

The `in` operator (`__contains__`) now only returns `True` for exact matches to *Intervals* in the *IntervalIndex*, whereas this would previously return `True` for any *Interval* overlapping an *Interval* in the *IntervalIndex*.

Previous behavior :

```
In [4]: pd.Interval(1, 2, closed='neither') in ii Out[4]: True
```

```
In [5]: pd.Interval(-10, 10, closed='both') in ii
```

```
Out[5]: True
```

New behavior:

```
In [41]: pd.Interval(1, 2, closed='neither') in ii
```

```
Out[41]: False
```

```
In [42]: pd.Interval(-10, 10, closed='both') in ii
```

```
Out[42]: False
```

The `get_loc()` method now only returns locations for exact matches to Interval queries, as opposed to the previous behavior of returning locations for overlapping matches. A `KeyError` will be raised if an exact match is not found.

Previous behavior :

```
In [6]: ii.get_loc(pd.Interval(1, 5))
```

```
Out[6]: array([0, 1])
```

```
In [7]: ii.get_loc(pd.Interval(2, 6))
```

```
Out[7]: array([0, 1, 2])
```

5.4. Version 0.25

New behavior:

```
In [6]: ii.get_loc(pd.Interval(1, 5))
```

```
Out[6]: 1
```

```
In [7]: ii.get_loc(pd.Interval(2, 6))
```

```
-----  
KeyError: Interval(2, 6, closed='right')
```

Likewise, `get_indexer()` and `get_indexer_non_unique()` will also only return locations for exact matches to Interval queries, with -1 denoting that an exact match was not found. These indexing changes extend to querying a *Series* or *DataFrame* with an *IntervalIndex* index.

```
In [43]: s = pd.Series(list('abc'), index=ii)
```

```
In [44]: s
```

```
Out[44]:
```

```
(0, 4] a
```

```
(1, 5] b
```



```
(5, 8] c
Length: 3, dtype: object
```

Selecting from a Series or DataFrame using `[]` (`__getitem__`) or `loc` now only returns exact matches for Interval queries.

Previous behavior :

```
In [8]: s[pd.Interval(1, 5)] Out[8]:
(0, 4] a
(1, 5] b
dtype: object
In [9]: s.loc[pd.Interval(1, 5)] Out[9]:
(0, 4] a
(1, 5] b
dtype: object
```

New behavior:

```
In [45]: s[pd.Interval(1, 5)]
Out[45]: 'b'
In [46]: s.loc[pd.Interval(1, 5)]
Out[46]: 'b'
```

Similarly, a `KeyError` will be raised for non-exact matches instead of returning overlapping matches.

Previous behavior :

```
In [9]: s[pd.Interval(2, 3)] Out[9]:
(0, 4] a
(1, 5] b
dtype: object
In [10]: s.loc[pd.Interval(2, 3)]
(continues on next page)
(continued from previous page)
Out[10]:
(0, 4] a
(1, 5] b
dtype: object
```

New behavior:

```
In [6]: s[pd.Interval(2, 3)]
```

KeyError: Interval(2, 3, closed='right')
In [7]: s.loc[pd.Interval(2, 3)]

KeyError: Interval(2, 3, closed='right')

The *overlaps()* method can be used to create a boolean indexer that replicates the previous behavior of returning overlapping matches.

New behavior:

In [47]: idxr = s.index.overlaps(pd.Interval(2, 3))

In [48]: idxr

Out[48]: array([True, True, False])

In [49]: s[idxr] **Out[49]:**

(0, 4] a

(1, 5] b

Length: 2, dtype: object

In [50]: s.loc[idxr] **Out[50]:**

(0, 4] a

(1, 5] b

Length: 2, dtype: object

Binary ufuncs on Series now align

Applying a binary ufunc like `numpy.power()` now aligns the inputs when both are *Series* (GH23293).

In [51]: s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c']) **In [52]:** s2 = pd.Series([3, 4, 5], index=['d', 'c', 'b'])

In [53]: s1 **Out[53]:** a 1 b 2 c 3

Length: 3, dtype: int64

In [54]: s2 **Out[54]:**

d 3

(continues on next page)

5.4. Version 0.25

(continued from previous page) c 4 b 5

Length: 3, dtype: int64

Previous behavior

In [5]: np.power(s1, s2)
Out[5]: a 1 b 16 c 243 dtype: int64
New behavior

In [55]: np.power(s1, s2)
Out[55]:
a 1.0 b 32.0 c 81.0
d NaN

Length: 4, dtype: float64

This matches the behavior of other binary operations in , like *Series.add()*. To retain the previous behavior, convert the other Series to an array before applying the ufunc.

In [56]: np.power(s1, s2.array) **Out[56]:** a 1 b 16 c 243
Length: 3, dtype: int64

Categorical.argsort now places missing values at the end

Categorical.argsort() now places missing values at the end of the array, making it consistent with NumPy and the rest of (GH21801).

In [57]: cat = pd.Categorical(['b', **None**, 'a'], categories=['a', 'b'],
ordered=**True**) *Previous behavior*

In [2]: cat = pd.Categorical(['b', **None**, 'a'], categories=['a', 'b'],
ordered=**True**) **In [3]:** cat.argsort()

Out[3]: array([1, 2, 0])

In [4]: cat[cat.argsort()] **Out[4]:**
[NaN, a, b] categories (2, object): [a < b]

New behavior

In [58]: cat.argsort()

Out[58]: array([2, 0, 1])

In [59]: cat[cat.argsort()] **Out[59]:**
['a', 'b', NaN]

Categories (2, object): ['a' < 'b']

Column order is preserved when passing a list of dicts to DataFrame

Starting with Python 3.7 the key-order of dict is guaranteed. In practice, this has been true since Python 3.6. The *DataFrame* constructor now treats a list of dicts in the same way as it does a list of OrderedDict, i.e. preserving the

order of the dicts. This change applies only when is running on Python>=3.6 (GH27309).

```
In [60]: data = [  
....: {'name': 'Joe', 'state': 'NY', 'age': 18},  
....: {'name': 'Jane', 'state': 'KY', 'age': 19, 'hobby': 'Minecraft'},  
  
....: {'name': 'Jean', 'state': 'OK', 'age': 20, 'finances': 'good'} ....: ] ....: Previous Behavior:
```

The columns were lexicographically sorted previously,

```
In [1]: pd.DataFrame(data) Out[1]:
```

```
age finances hobby name state  
18 NaN NaN Joe NY  
19 NaN Minecraft Jane KY  
20 good NaN Jean OK  
New Behavior:
```

The column order now matches the insertion-order of the keys in the dict, considering all the records from top to bottom. As a consequence, the column order of the resulting DataFrame has changed compared to previous versions.

```
In [61]: pd.DataFrame(data) Out[61]:
```

```
name state age hobby finances  
Joe NY 18 NaN NaN  
Jane KY 19 Minecraft NaN  
Jean OK 20 NaN good  
[3 rows x 5 columns]
```

5.4. Version 0.25

Increased minimum versions for dependencies

Due to dropping support for Python 2.7, a number of optional dependencies have updated minimum versions (GH25725, GH24942, GH25752).

Independently, some minimum supported versions of dependencies were updated (GH23519, GH25554). If installed, we now require:

Package Minimum Required

Version

numpy 1.13.3 X pytz 2015.4 X

python2.6.1 X dateutil

bottleneck 1.2.1

numexpr 2.6.2

pytest 4.0.2

(dev)

For optional libraries the general recommendation is to use the latest version. The following table lists the lowest version per library that is currently being tested throughout the development of . Optional libraries below the lowest tested version may still work, but are not considered supported.

Package Minimum Version beautifulsoup4 4.6.0 fastparquet 0.2.1 gcsfs 0.2.2
lxml 3.8.0 matplotlib 2.2.2 openpyxl 2.4.8 pyarrow 0.9.0 pymysql 0.7.1
pytables 3.4.2 scipy 0.19.0 sqlalchemy 1.1.4 xarray 0.8.2 xlrd 1.1.0
xlsxwriter 0.9.8 xlwt 1.2.0

See *Dependencies* and *Optional dependencies* for more.

Other API changes

DatetimeTZDtype will now standardize pytz timezones to a common timezone instance (GH24713)

Timestamp and *Timedelta* scalars now implement the `to_numpy()` method as aliases to *Timestamp.to_datetime64()* and *Timedelta.to_timedelta64()*, respectively. (GH24653) *Timestamp.strptime()* will now raise a `NotImplementedError` (GH25016)

Comparing *Timestamp* with unsupported objects now returns `NotImplemented` instead of raising `TypeError`. This implies that unsupported rich comparisons are delegated to the other object, and are now consistent with Python 3 behavior for datetime objects (GH24011)

Bug in *DatetimeIndex.snap()* which didn't preserving the name of the input *Index* (GH25575)

The `arg` argument in `.core.groupby.DataFrameGroupBy.agg()` has been renamed to `func` (GH26089)

The `arg` argument in `.core.window._Window.aggregate()` has been renamed to `func` (GH26372)

Most classes had a `__bytes__` method, which was used for getting a python2-style bytestring representation of the object. This method has been removed as a part of dropping Python2 (GH26447)

The `.str`-accessor has been disabled for 1-level *MultiIndex*, use *MultiIndex.to_flat_index()* if necessary (GH23679)

Removed support of `gtk` package for clipboards (GH26563)

Using an unsupported version of Beautiful Soup 4 will now raise an `ImportError` instead of a `ValueError` (GH27063)

Series.to_excel() and *DataFrame.to_excel()* will now raise a `ValueError` when saving timezone aware data. (GH27008, GH7056)

ExtensionArray.argsort() places NA values at the end of the sorted array. (GH21801)

DataFrame.to_hdf() and *Series.to_hdf()* will now raise a `NotImplementedError` when saving a *MultiIndex* with extension data types for a fixed format. (GH7775)

Passing duplicate names in *read_csv()* will now raise a `ValueError` (GH17346) **Deprecations**

Sparse subclasses

The *SparseSeries* and *SparseDataFrame* subclasses are deprecated. Their functionality is better-provided by a *Series* or *DataFrame* with sparse values. Previous way

```
df = pd.SparseDataFrame({"A": [0, 0, 1, 2]}) df.dtypes
```

New way

```
In [62]: df = pd.DataFrame({"A": pd.SparseArray([0, 0, 1, 2])})
```

```
In [63]: df.dtypes Out[63]:
```

```
A Sparse[int64, 0]
```

```
Length: 1, dtype: object
```

The memory usage of the two approaches is identical. See *Migrating* for more (GH19239).

msgpack format

The msgpack format is deprecated as of 0.25 and will be removed in a future version. It is recommended to use pyarrow for on-the-wire transmission of objects. (GH27084)

5.4. Version 0.25

Other deprecations

The deprecated `.ix[]` indexer now raises a more visible `FutureWarning` instead of `DeprecationWarning` (GH26438).

Deprecated the `units=M` (months) and `units=Y` (year) parameters for units of `.to_timedelta()`, `.Timedelta()` and `.TimedeltaIndex()` (GH16344)

`.concat()` has deprecated the `join_axes`-keyword. Instead, use `DataFrame.reindex()` or `DataFrame.reindex_like()` on the result or on the inputs (GH21951)

The `SparseArray.values` attribute is deprecated. You can use `np.asarray(...)` or the `SparseArray.to_dense()` method instead (GH26421).

The functions `.to_datetime()` and `.to_timedelta()` have deprecated the `box` keyword. Instead, use `to_numpy()` or `Timestamp.to_datetime64()` or `Timedelta.to_timedelta64()`. (GH24416)

The `DataFrame.compound()` and `Series.compound()` methods are deprecated and will be removed in a future version (GH26405).

The internal attributes `_start`, `_stop` and `_step` attributes of `RangeIndex` have been deprecated. Use the public attributes `start`, `stop` and `step` instead (GH26581).

The `Series.ftype()`, `Series.ftypes()` and `DataFrame.ftypes()` methods are deprecated and will be removed in a future version. Instead, use `Series.dtype()` and `DataFrame.dtypes()` (GH26705).

The `Series.get_values()`, `DataFrame.get_values()`, `Index.get_values()`, `SparseArray.get_values()` and `Categorical.get_values()` methods are deprecated. One of `np.asarray(..)` or `to_numpy()` can be used instead (GH19617).

The ‘outer’ method on NumPy ufuncs, e.g. `np.subtract.outer` has been deprecated on `Series` objects. Convert the input to an array with `Series.array`

first (GH27186)

Timedelta.resolution() is deprecated and replaced with *Timedelta.resolution_string()*. In a future version, *Timedelta.resolution()* will be changed to behave like the standard library `datetime.timedelta.resolution` (GH21344)

read_table() has been undeprecated. (GH25220)

Index.dtype_str is deprecated. (GH18262)

Series.imag and *Series.real* are deprecated. (GH18262)

Series.put() is deprecated. (GH18262)

Index.item() and *Series.item()* is deprecated. (GH18262)

The default value `ordered=None` in *CategoricalDtype* has been deprecated in favor of `ordered=False`. When converting between categorical types `ordered=True` must be explicitly passed in order to be preserved. (GH26336)

Index.contains() is deprecated. Use `key in index` (`__contains__`) instead (GH17753). *DataFrame.get_dtype_counts()* is deprecated. (GH18262)

Categorical.ravel() will return a *Categorical* instead of a `np.ndarray`

(GH27199) **Removal of prior version deprecations/changes**

Removed Panel (GH25047, GH25191, GH25231)

Removed the previously deprecated `sheetname` keyword in *read_excel()* (GH16442, GH20938)

Removed the previously deprecated *TimeGrouper* (GH16942)

Removed the previously deprecated `parse_cols` keyword in *read_excel()*

(GH16488) Removed the previously deprecated `pd.options.html.border`

(GH16970) Removed the previously deprecated `convert_objects` (GH11221)

Removed the previously deprecated `select` method of *DataFrame* and *Series* (GH17633)

Removed the previously deprecated behavior of *Series* treated as list-like in *rename_categories()* (GH17982)

Removed the previously deprecated *DataFrame.reindex_axis* and *Series.reindex_axis* (GH17842)

Removed the previously deprecated behavior of altering column or index labels with *Series.rename_axis()* or *DataFrame.rename_axis()* (GH17842)

Removed the previously deprecated *tupleize_cols* keyword argument in *read_html()*, *read_csv()*, and *DataFrame.to_csv()* (GH17877, GH17820)

Removed the previously deprecated *DataFrame.from_csv* and *Series.from_csv* (GH17812)

Removed the previously deprecated *raise_on_error* keyword argument in *DataFrame.where()* and *DataFrame.mask()* (GH17744)

Removed the previously deprecated *ordered* and *categories* keyword arguments in *astype* (GH17742)

Removed the previously deprecated *cdate_range* (GH17691)

Removed the previously deprecated *True* option for the *dropna* keyword argument in *SeriesGroupBy.nth()* (GH17493)

Removed the previously deprecated *convert* keyword argument in *Series.take()* and *DataFrame.take()* (GH17352)

Removed the previously deprecated behavior of arithmetic operations with *datetime.date* objects (GH21152)

Performance improvements

Significant speedup in *SparseArray* initialization that benefits most operations, fixing performance regression introduced in v0.20.0 (GH24985)
DataFrame.to_stata() is now faster when outputting data with any string or non-native endian columns (GH25045)

Improved performance of *Series.searchsorted()*. The speedup is especially large when the *dtype* is *int8/int16/int32* and the searched key is within the integer bounds for the *dtype* (GH22034)

Improved performance of *.core.groupby.GroupBy.quantile()* (GH20405)

Improved performance of slicing and other selected operation on a *RangeIndex* (GH26565, GH26617, GH26722)

RangeIndex now performs standard lookup without instantiating an actual hashtable, hence saving memory

(GH16685)

5.4. Version 0.25

Improved performance of *read_csv()* by faster tokenizing and faster parsing

of small float numbers (GH25784)

Improved performance of *read_csv()* by faster parsing of N/A and boolean values (GH25804)

Improved performance of *IntervalIndex.is_monotonic*, *IntervalIndex.is_monotonic_increasing* and *IntervalIndex.is_monotonic_decreasing* by removing conversion to *MultiIndex* (GH24813)

Improved performance of *DataFrame.to_csv()* when writing datetime dtypes (GH25708) Improved performance of *read_csv()* by much faster parsing of MM/YYYY and DD/MM/YYYY datetime formats (GH25922)

Improved performance of nanops for dtypes that cannot store NaNs. Speedup is particularly prominent for *Series.all()* and *Series.any()* (GH25070)

Improved performance of *Series.map()* for dictionary mappers on categorical series by mapping the categories instead of mapping all values (GH23785)

Improved performance of *IntervalIndex.intersection()* (GH24813)

Improved performance of *read_csv()* by faster concatenating date columns without extra conversion to string for integer/float zero and float NaN; by faster checking the string for the possibility of being a date (GH25754)

Improved performance of *IntervalIndex.is_unique* by removing conversion to *MultiIndex* (GH24813)

Restored performance of *DatetimeIndex.__iter__()* by re-enabling specialized code path (GH26702)

Improved performance when building *MultiIndex* with at least one *CategoricalIndex* level (GH22044)

Improved performance by removing the need for a garbage collect when checking for *SettingWithCopyWarning* (GH27031) For *to_datetime()* changed default value of *cache* parameter to *True* (GH26043) Improved performance of *DatetimeIndex* and *PeriodIndex* slicing given non-unique, monotonic data (GH27136).

Improved performance of *pd.read_json()* for index-oriented data. (GH26773)

Improved performance of *MultiIndex.shape()* (GH27384).

Bug fixes

Categorical

Bug in *DataFrame.at()* and *Series.at()* that would raise exception if the index was a *CategoricalIndex* (GH20629)

Fixed bug in comparison of ordered *Categorical* that contained missing values with a scalar which sometimes incorrectly resulted in True (GH26504)

Bug in *DataFrame.dropna()* when the *DataFrame* has a *CategoricalIndex* containing *Interval* objects incorrectly raised a *TypeError* (GH25087)

Datetimelike

Bug in *to_datetime()* which would raise an (incorrect) *ValueError* when called with a date far into the future and the format argument specified instead of raising *OutOfBoundsDatetime* (GH23830)

Bug in *to_datetime()* which would raise *InvalidIndexError*: Reindexing only valid with uniquely valued Index objects when called with *cache=True*, with arg including at least two different elements from the set {None, numpy.nan, .NaT} (GH22305)

Bug in *DataFrame* and *Series* where timezone aware data with *dtype='datetime64[ns]'* was not cast to naive (GH25843)

Improved *Timestamp* type checking in various datetime functions to prevent exceptions when using a subclassed datetime (GH25851)

Bug in *Series* and *DataFrame* repr where *np.datetime64('NaT')* and *np.timedelta64('NaT')* with *dtype=object* would be represented as *NaN* (GH25445) Bug in *to_datetime()* which does not replace the invalid argument with *NaT* when error is set to *coerce* (GH26122)

Bug in adding *DateOffset* with nonzero month to *DatetimeIndex* would raise *ValueError* (GH26258)

Bug in *to_datetime()* which raises unhandled *OverflowError* when called with mix of invalid dates and *NaN* values with *format='%Y%m%d'* and *error='coerce'* (GH25512) Bug in *isin()* for datetimelike indexes;

DatetimeIndex, *TimedeltaIndex* and *PeriodIndex* where the *levels* parameter was ignored. (GH26675)

Bug in *to_datetime()* which raises *TypeError* for *format='%Y%m%d'* when called for invalid integer dates with length ≥ 6 digits with *errors='ignore'*

Bug when comparing a *PeriodIndex* against a zero-dimensional numpy array (GH26689)

Bug in constructing a *Series* or *DataFrame* from a numpy *datetime64* array

with a nonns unit and outof-bound timestamps generating rubbish data, which will now correctly raise an OutOfBoundsDatetime error (GH26206).

Bug in *date_range()* with unnecessary OverflowError being raised for very large or very small dates (GH26651)

Bug where adding *Timestamp* to a np.timedelta64 object would raise instead of returning a *Timestamp* (GH24775)

Bug where comparing a zero-dimensional numpy array containing a np.datetime64 object to a *Timestamp* would incorrect raise TypeError (GH26916)

Bug in *to_datetime()* which would raise ValueError: Tz-aware datetime.datetime cannot be converted to datetime64 unless utc=True when called with cache=True, with arg including datetime strings with different offset (GH26097)

-

5.4. Version 0.25

Timedelta

Bug in TimedeltaIndex.intersection() where for non-monotonic indices in some cases an empty Index was returned when in fact an intersection existed (GH25913) Bug with comparisons between *Timedelta* and NaT raising TypeError (GH26039) Bug when adding or subtracting a BusinessHour to a *Timestamp* with the resulting time landing in a following or prior day respectively (GH26381)

Bug when comparing a *TimedeltaIndex* against a zero-dimensional numpy array (GH26689)

Timezones

Bug in *DatetimeIndex.to_frame()* where timezone aware data would be converted to timezone naive data (GH25809)

Bug in *to_datetime()* with utc=True and datetime strings that would apply previously parsed UTC offsets to subsequent arguments (GH24992)

Bug in *Timestamp.tz_localize()* and *Timestamp.tz_convert()* does not propagate freq (GH25241)

Bug in *Series.at()* where setting *Timestamp* with timezone raises TypeError (GH25506)

Bug in *DataFrame.update()* when updating with timezone aware

data would return timezone naive data (GH25807)

Bug in `to_datetime()` where an uninformative `RuntimeError` was raised when passing a naive *Timestamp* with datetime strings with mixed UTC offsets

(GH25978) Bug in `to_datetime()` with `unit='ns'` would drop timezone information from the parsed argument (GH26168)

Bug in `DataFrame.join()` where joining a timezone aware index with a timezone aware column would result in a column of NaN (GH26335)

Bug in `date_range()` where ambiguous or nonexistent start or end times were not handled by the ambiguous or nonexistent keywords respectively

(GH27088) Bug in `DatetimeIndex.union()` when combining a timezone aware and timezone unaware *DatetimeIndex* (GH21671)

Bug when applying a numpy reduction function (e.g. `numpy.minimum()`) to a timezone aware *Series*

(GH15552)

Numeric

Bug in `to_numeric()` in which large negative numbers were being improperly handled (GH24910)

Bug in `to_numeric()` in which numbers were being coerced to float, even though errors was not coerce (GH24910)

Bug in `to_numeric()` in which invalid values for errors were being allowed (GH26466) Bug in `format` in which floating point complex numbers were not being formatted to proper display precision and trimming (GH25514)

Bug in error messages in `DataFrame.corr()` and `Series.corr()`. Added the possibility of using a callable. (GH25729)

Bug in `Series.divmod()` and `Series.rdivmod()` which would raise an (incorrect) `ValueError` rather than return a pair of *Series* objects as result (GH25557)

Raises a helpful exception when a non-numeric index is sent to `interpolate()` with methods which require numeric index. (GH21662)

Bug in `eval()` when comparing floats with scalar operators, for example: `x < -0.1` (GH25928)

Fixed bug where casting all-boolean array to integer extension array failed

(GH25211) Bug in `divmod` with a *Series* object containing zeros incorrectly raising `AttributeError` (GH26987)

Inconsistency in *Series* floor-division (`//`) and `divmod` filling positive`//zero`

with NaN instead of Inf (GH27321)

-

Conversion

Bug in *DataFrame.astype()* when passing a dict of columns and types the errors parameter was ignored. (GH25905)

-

Strings

Bug in the `__name__` attribute of several methods of *Series.str*, which were set incorrectly (GH23551)

Improved error message when passing *Series* of wrong dtype to *Series.str.cat()* (GH22722)

-

Interval

Construction of *Interval* is restricted to numeric, *Timestamp* and *Timedelta* endpoints (GH23013)

Fixed bug in *Series/DataFrame* not displaying NaN in *IntervalIndex* with missing values (GH25984)

Bug in *IntervalIndex.get_loc()* where a *KeyError* would be incorrectly raised for a decreasing *IntervalIndex* (GH25860)

Bug in *Index* constructor where passing mixed closed *Interval* objects would result in a *ValueError* instead of an object dtype *Index* (GH27172)

Indexing

Improved list exception message when calling *DataFrame.iloc()* with a of non-numeric objects (GH25753).

Improved exception message when calling *.iloc* or *.loc* with a boolean indexer with different length (GH26658).

Bug in *KeyError* exception message when indexing a *MultiIndex* with a non-existent key not displaying the original key (GH27250).

Bug in *.iloc* and *.loc* with a boolean indexer not raising an *IndexError* when too few items are passed (GH26658).

5.4. Version 0.25

Bug in *DataFrame.loc()* and *Series.loc()* where *KeyError* was not raised for a *MultiIndex* when the key was less than or equal to the number of levels in the *MultiIndex* (GH14885).

Bug in which *DataFrame.append()* produced an erroneous warning indicating that a *KeyError* will be thrown in the future when the data to be appended contains new columns (GH22252).

Bug in which *DataFrame.to_csv()* caused a segfault for a reindexed data frame, when the indices were single-level *MultiIndex* (GH26303).

Fixed bug where assigning a *arrays.Array* to a *.core.frame.DataFrame* would raise error (GH26390)

Allow keyword arguments for callable local reference used in the *DataFrame.query()* string (GH26426)

Fixed a *KeyError* when indexing a *MultiIndex`* level with a list containing exactly one label, which is missing (GH27148)

Bug which produced *AttributeError* on partial matching *Timestamp* in a *MultiIndex* (GH26944)

Bug in *Categorical* and *CategoricalIndex* with *Interval* values when using the *in* operator (*__contains__*) with objects that are not comparable to the values in the *Interval* (GH23705)

Bug in *DataFrame.loc()* and *DataFrame.iloc()* on a *DataFrame* with a single *timezoneaware datetime64[ns]* column incorrectly returning a scalar instead of a *Series* (GH27110)

Bug in *CategoricalIndex* and *Categorical* incorrectly raising *ValueError* instead of *TypeError* when a list is passed using the *in* operator

(*__contains__*) (GH21729) Bug in setting a new value in a *Series* with a *Timedelta* object incorrectly casting the value to an integer (GH22717)

Bug in *Series* setting a new key (*__setitem__*) with a *timezone-aware datetime* incorrectly raising *ValueError* (GH12862)

Bug in *DataFrame.iloc()* when indexing with a read-only indexer (GH17192)

Bug in *Series* setting an existing tuple key (*__setitem__*) with *timezone-aware datetime* values incorrectly raising *TypeError* (GH20441)

Missing

Fixed misleading exception message in *Series.interpolate()* if argument order

is required, but omitted (GH10633, GH24014).

Chapter 5. Release notes

Fixed class type displayed in exception message in *DataFrame.dropna()* if invalid axis parameter passed (GH25555)

A *ValueError* will now be thrown by *DataFrame.fillna()* when limit is not a positive integer (GH27042)

-

2704

MultiIndex

Bug in which incorrect exception raised by *Timedelta* when testing the membership of *MultiIndex* (GH24570)

-

IO

Bug in *DataFrame.to_html()* where values were truncated using display options instead of outputting the full content (GH17004)

Fixed bug in missing text when using *to_clipboard()* if copying utf-16 characters in Python 3 on Windows (GH25040)

Bug in *read_json()* for *orient='table'* when it tries to infer dtypes by default, which is not applicable as dtypes are already defined in the JSON schema (GH21345)

Bug in *read_json()* for *orient='table'* and float index, as it infers index dtype by default, which is not applicable because index dtype is already defined in the JSON schema (GH25433)

Bug in *read_json()* for *orient='table'* and string of float column names, as it makes a column name type conversion to *Timestamp*, which is not applicable because column names are already defined in the JSON schema (GH25435)

Bug in *json_normalize()* for *errors='ignore'* where missing values in the input data, were filled in resulting *DataFrame* with the string "nan" instead of *numpy.nan* (GH25468) *DataFrame.to_html()* now raises *TypeError* when using an invalid type for the *classes* parameter instead of *AssertionError*

(GH25608)

Bug in *DataFrame.to_string()* and *DataFrame.to_latex()* that would lead to incorrect output when the header keyword is used (GH16718)

Bug in *read_csv()* not properly interpreting the UTF8 encoded filenames on Windows on Python 3.6+ (GH15086)

Improved performance in *.read_stata()* and *.io.stata.StataReader* when converting columns that have missing values (GH25772)

Bug in *DataFrame.to_html()* where header numbers would ignore display options when rounding (GH17280)

Bug in *read_hdf()* where reading a table from an HDF5 file written directly with PyTables fails with a *ValueError* when using a sub-selection via the start or stop arguments (GH11188)

Bug in *read_hdf()* not properly closing store after a *KeyError* is raised (GH25766) Improved the explanation for the failure when value labels are repeated in Stata dta files and suggested workarounds (GH25772)

Improved *.read_stata()* and *.io.stata.StataReader* to read incorrectly formatted 118 format files saved by Stata (GH25960)

Improved the *col_space* parameter in *DataFrame.to_html()* to accept a string so CSS length values can be set correctly (GH25941)

Fixed bug in loading objects from S3 that contain # characters in the URL (GH25945)

Adds *use_bqstorage_api* parameter to *read_gbq()* to speed up downloads of large data frames. This feature requires version 0.10.0 of the *-gbq* library as well as the *googlecloud-bigquery-storage* and *fastavro* libraries. (GH26104)

5.4. Version 0.25

Fixed memory leak in *DataFrame.to_json()* when dealing with numeric data (GH24889)

Bug in *read_json()* where date strings with Z were not converted to a UTC timezone (GH26168)

Added *cache_dates=True* parameter to *read_csv()*, which allows to cache unique dates when they are parsed (GH25990)

DataFrame.to_excel() now raises a *ValueError* when the caller's dimensions

exceed the limitations of Excel (GH26051)

Fixed bug in *.read_csv()* where a BOM would result in incorrect parsing using engine='python' (GH26545)

read_excel() now raises a ValueError when input is of type

.io.excel.ExcelFile and engine param is passed since *.io.excel.ExcelFile* has an engine defined (GH26566) Bug while selecting from HDFStore with where="" specified (GH26610).

Fixed bug in *DataFrame.to_excel()* where custom objects (i.e. *PeriodIndex*) inside merged cells were not being converted into types safe for the Excel writer (GH27006)

Bug in *read_hdf()* where reading a timezone aware *DatetimeIndex* would raise a TypeError (GH11926)

Bug in *to_msgpack()* and *read_msgpack()* which would raise a ValueError rather than a FileNotFoundError for an invalid path (GH27160)

Fixed bug in *DataFrame.to_parquet()* which would raise a ValueError when the dataframe had no columns (GH27339)

Allow parsing of *PeriodDtype* columns when using *read_csv()* (GH26934)

Plotting

Fixed bug where *api.extensions.ExtensionArray* could not be used in matplotlib plotting (GH25587)

Bug in an error message in *DataFrame.plot()*. Improved the error message if nonnumerics are passed to *DataFrame.plot()* (GH25481)

Bug in incorrect ticklabel positions when plotting an index that are non-numeric / nondatetime (GH7612, GH15912, GH22334)

Fixed bug causing plots of *PeriodIndex* timeseries to fail if the frequency is a multiple of the frequency rule code (GH14763)

Fixed bug when plotting a *DatetimeIndex* with *datetime.timezone.utc* timezone (GH17173)

•

GroupBy/resample/rolling

Bug in *.core.resample.Resampler.agg()* with a timezone aware index where *OverflowError* would raise when passing a list of functions (GH22660)

Bug in `.core.groupby.DataFrameGroupBy.nunique()` in which the names of column levels were lost (GH23222)

Bug in `.core.groupby.GroupBy.agg()` when applying an aggregation function to timezone aware data (GH23683)

Bug in `.core.groupby.GroupBy.first()` and `.core.groupby.GroupBy.last()` where timezone information would be dropped (GH21603)

Bug in `.core.groupby.GroupBy.size()` when grouping only NA values (GH23050) Bug in `Series.groupby()` where observed kwarg was previously ignored (GH24880)

Bug in `Series.groupby()` where using groupby with a *MultiIndex* Series with a list of labels equal to the length of the series caused incorrect grouping (GH25704)

Ensured that ordering of outputs in groupby aggregation functions is consistent across all versions of Python (GH25692)

Ensured that result group order is correct when grouping on an ordered Categorical and specifying `observed=True` (GH25871, GH25167)

Bug in `.core.window.Rolling.min()` and `.core.window.Rolling.max()` that caused a memory leak (GH25893)

Bug in `.core.window.Rolling.count()` and `.core.window.Expanding.count` was previously ignoring the `axis` keyword (GH13503)

Bug in `.core.groupby.GroupBy.idxmax()` and `.core.groupby.GroupBy.idxmin()` with datetime column would return incorrect dtype (GH25444, GH15306)

Bug in `.core.groupby.GroupBy.cumsum()`, `.core.groupby.GroupBy.cumprod()`, `.core.groupby.GroupBy.cummin()` and `.core.groupby.GroupBy.cummax()` with categorical column having absent categories, would return incorrect result or segfault (GH16771)

Bug in `.core.groupby.GroupBy.nth()` where NA values in the grouping would return incorrect results (GH26011)

Bug in `.core.groupby.SeriesGroupBy.transform()` where transforming an empty group would raise a `ValueError` (GH26208)

Bug in `.core.frame.DataFrame.groupby()` where passing a `.core.groupby.grouper.Grouper` would return incorrect groups when using the `.groups`

accessor (GH26326)

Bug in `.core.groupby.GroupBy.agg()` where incorrect results are returned for uint64 columns. (GH26310)

Bug in `.core.window.Rolling.median()` and `.core.window.Rolling.quantile()` where `MemoryError` is raised with empty window (GH26005)

Bug in `.core.window.Rolling.median()` and `.core.window.Rolling.quantile()` where incorrect results are returned with `closed='left'` and `closed='neither'` (GH26005)

Improved `.core.window.Rolling`, `.core.window.Window` and `.core.window.ExponentialMovingWindow` functions to exclude nuisance columns from results instead of raising errors and raise a `DataError` only if all columns are nuisance (GH12537)

Bug in `.core.window.Rolling.max()` and `.core.window.Rolling.min()` where incorrect results are returned with an empty variable window (GH26005)

Raise a helpful exception when an unsupported weighted window function is used as an argument of `.core.window.Window.aggregate()` (GH26597)

5.4. Version 0.25

Reshaping

Bug in `.merge()` adds a string of `None`, if `None` is assigned in suffixes instead of remain the column name as-is (GH24782).

Bug in `.merge()` when merging by index name would sometimes result in an incorrectly numbered index

(missing index values are now assigned `NA`) (GH24212, GH25009)

`to_records()` now accepts dtypes to its `column_dtypes` parameter (GH24895)

Bug in `concat()` where order of `OrderedDict` (and `dict` in Python 3.6+) is not respected, when passed in as `objs` argument (GH21510)

Bug in `pivot_table()` where columns with `NaN` values are dropped even if `dropna` argument is `False`, when the `aggfunc` argument contains a list (GH22159) Bug in `concat()` where the resulting freq of two `DatetimeIndex` with the same freq would be dropped (GH3232).

Bug in `.merge()` where merging with equivalent Categorical dtypes was raising an error (GH22501)

bug in `DataFrame` instantiating with a dict of iterators or generators (e.g.

`pd.DataFrame({'A': reversed(range(3))})` raised an error (GH26349).

Bug in *DataFrame* instantiating with a range (e.g. `pd.DataFrame(range(3))`) raised an error (GH26342).

Bug in *DataFrame* constructor when passing non-empty tuples would cause a segmentation fault (GH25691)

Bug in *Series.apply()* failed when the series is a timezone aware *DatetimeIndex* (GH25959)

Bug in *.cut()* where large bins could incorrectly raise an error due to an integer overflow (GH26045)

Bug in *DataFrame.sort_index()* where an error is thrown when a multi-indexed *DataFrame* is sorted on all levels with the initial level sorted last (GH26053) Bug in *Series.nlargest()* treats True as smaller than False (GH26154)

Bug in *DataFrame.pivot_table()* with a *IntervalIndex* as pivot index would raise *TypeError* (GH25814)

Bug in which *DataFrame.from_dict()* ignored order of *OrderedDict* when `orient='index'` (GH8425).

Bug in *DataFrame.transpose()* where transposing a *DataFrame* with a timezone-aware datetime column would incorrectly raise *ValueError* (GH26825)

Bug in *pivot_table()* when pivoting a timezone aware column as the values would remove timezone information (GH14948)

Bug in *merge_asof()* when specifying multiple by columns where one is `datetime64[ns, tz]` dtype

(GH26649)

Sparse

Significant speedup in *SparseArray* initialization that benefits most operations, fixing performance regression introduced in v0.20.0 (GH24985)

Bug in *SparseFrame* constructor where passing None as the data would cause `default_fill_value` to be ignored (GH16807)

Bug in *SparseDataFrame* when adding a column in which the length of values does not match length of index, *AssertionError* is raised instead of raising *ValueError* (GH25484) Introduce a better error message in

Series.sparse.from_coo() so it returns a `TypeError` for inputs that are not coo matrices (GH26554)

Bug in `numpy.modf()` on a `SparseArray`. Now a tuple of `SparseArray` is returned (GH26946).

Build changes

Fix install error with PyPy on macOS (GH26536)

ExtensionArray

Bug in *factorize()* when passing an `ExtensionArray` with a custom `na_sentinel` (GH25696).

Series.count() miscounts NA values in `ExtensionArrays` (GH26835)

Added `Series.__array_ufunc__` to better handle NumPy ufuncs applied to Series backed by extension arrays (GH23293).

Keyword argument `deep` has been removed from `ExtensionArray.copy()` (GH27083) **Other**

Removed unused C functions from vendored UltraJSON implementation (GH26198) Allow *Index* and *RangeIndex* to be passed to numpy min and max functions (GH26125) Use actual class name in repr of empty objects of a Series subclass (GH27001).

Bug in *DataFrame* where passing an object array of timezone-aware datetime objects would incorrectly raise `ValueError` (GH13287)

Contributors

A total of 231 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

1_x7 +

Abdullah İhsan Seçer +

Adam Bull +

Adam Hooper

Albert Villanova del Moral

Alex Watt +

5.4. Version 0.25

-
-

AlexTereshenkov + Alexander
Buchkovsky

Alexander Hendorf + Alexander Nordin +

Alexander
Ponomaroff

Alexandre Batisse + Alexandre Decan + Allen Downey +
Alyssa Fu Ward + Andrew Gaspari + Andrew Wood +
Antoine Viscardi + Antonio Gutierrez + Arno Veenstra +
ArtinSarraf
Batalex +

Baurzhan
Muftakhidinov
Benjamin Rowell Bharat Raghunathan +

-
-
-
-

-

Bhavani Ravi + • Big Head +

Brett Randall +
Bryan Cutler +
C John Klehm +
Caleb Braun +
Cecilia +
Chris Bertinato + Chris Stadler +
Christian Haege + Christian Hudon
Christopher Whelan

Chuanzhu Xu +
Clemens Brunner

Damian Kula +
Daniel Hrisca +
Daniel Luis Costa + Daniel Saxton
DanielFEvans +
David Liu +
Deepyaman Datta + Denis Belavin +
Devin Petersohn +

-
-
-
-

•
Diane Trout +

EdAbati +
Enrico Rotundo + EternalLearner42 + Evan +
Evan Livelo +
Fabian Rost +
Flavien Lambert + Florian Rathgeber + Frank Hoang +
Gaibo Zhang +
Gioia Ballin

Giuseppe
Romagnuolo +

Gordon Blackadder + Gregory Rome +
Guillaume Gay
HHest +
Hielke Walinga +
How Si Wei +
Hubert
Huize Wang +
Hyukjin Kwon +

-
-

-
-
-

Ian Dunn +

Inevitable-Marzipan + Irv Lustig

JElfner +

Jacob Bundgaard + James Cobon-Kerr + Jan-Philip Gehrcke + Jarrod

Millman +

Jayanth Katuri + Jeff Reback

Jeremy Schendel Jiang Yue +

Joel Ostblom

Johan von Forstner + Johnny Chiu +

Jonas +

Jonathon

Vandezande +

Jop Vermeer +

Joris Van den

Bossche

Josh

Josh Friedlander +

Justin Zheng

-
-
-
-

-

Kaiqi Dong

Kane +

Kapil Patel +

Kara de la Marck + Katherine Surta + Katrin Leinweber + Kendall Masse
Kevin Sheppard
Kyle Kosic +
Lorenzo Stella +
Maarten Rietbergen + Mak Sze Chun
Marc Garcia
Mateusz Wos´
Matias Heikkilä
Mats Maiwald +
Matthew Roeschke Max Bolingbroke + Max Kovalovs +
Max van Deursen + Michael
Michael Davis +

Michael P. Moran +

-
-
-
-

-

Mike Cramblett +

Min ho Kim +

Misha Veldhoen +

Mukul Ashwath Ram +

MusTheDataGuy + Nanda H Krishna + Nicholas Musolino Noam Hershtig +

Noora Hussein +

Paul

Paul Reidy

Pauli Virtanen

Pav A +

Peter Leimbögl +

Philippe Ombredanne +

Pietro Battiston

Richard Eames +
Roman Yurchak
Ruijing Li
Ryan
Ryan Joyce +

-
-
-
-

•
Ryan Nazareth

Ryan Rehman + Sakar Panta +
Samuel Sinayoko Sandeep Pathak + Sangwoong Yoon Saurav Chakravorty
Scott Talbert +
Sergey Kopylov + Shantanu Gontia + Shivam Rana +
Shorokhov Sergey + Simon Hawkins

Soyoun(Rose) Kim Stephan Hoyer

Stephen Cowley + Stephen Rauch
Sterling Paramore + Steven +
Stijn Van Hoey
Sumanau Sareen + Takuya N +
Tan Tran +

-
-
-
-

•
Tao He +

Tarbo Fukazawa
Terji Petersen +

Thein Oo
ThibTrip +
Thijs Damsma +

Thiviyan
Thanapalasingam

Thomas A Caswell Thomas Kluitters + Tilen Kusterle +
Tim Gates +
Tim Hoffmann
Tim Swast

Tom Augspurger • Tom Neep +
Tomáš Chvátal + • Tyler Reddy

Vaibhav Vishal +
Vasily Litvinov +
Vibhu Agarwal +
Vikramjeet Das +
Vladislav +

Víctor Moron Tejero +

-
-
-
-
-

Wenhuan

Will Ayd +
William Ayd
Wouter De Coster +

Yoann Goular + Zach Angell +
alimcmaster1
anmyachev +

chris-b1
danielplawrence + endenis +
enisnazif +
ezcitron +
fjetter
froessler
gfyong
gwrome +
h-vetinari
haison +
hannah-c +
heckeop +
iamshwin +
jamesoliverh +
jbrockmendel

•

•

•

•

•

jkovacevic +

killerontherun1 + knuu +
kpapdac +
kpflugshaupt +
krsnik93 +
leerssej +
lrjball +
mazayo +
nathalier +

nrebena + • nullptr + pilkibun + pmaxey83 + rbenes +
robbuckley

•

•

•

-

-

shawnbrown + sudhir mohanraj + tadeja + tamuhey +

thatneat

topper-123

willweil +

yehia67 +

yhaque1213 +

5.5 Version 0.24

5.5.1 What's new in 0.24.2 (March 12, 2019)

Warning: The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See [Dropping Python 2.7](#) for more. These are the changes in 0.24.2. See *Release notes* for a full changelog including other versions of .

Fixed regressions

Fixed regression in *DataFrame.all()* and *DataFrame.any()* where `bool_only=True` was ignored (GH25101)

Fixed issue in *DataFrame* construction with passing a mixed list of mixed types could segfault. (GH25075)

Fixed regression in *DataFrame.apply()* causing *RecursionError* when dict-like classes were passed as argument. (GH25196)

Fixed regression in *DataFrame.replace()* where `regex=True` was only replacing patterns matching the start of the string (GH25259)

Fixed regression in *DataFrame.duplicated()*, where empty dataframe was not returning a boolean dtyped Series. (GH25184)

Fixed regression in *Series.min()* and *Series.max()* where `numeric_only=True` was ignored when the Series contained Categorical data (GH25299)

Fixed regression in subtraction between *Series* objects with `datetime64[ns]` dtype incorrectly raising *OverflowError* when the Series on the right contains null values (GH25317)

-

711 Chapter 5. Release notes

Fixed regression in *TimedeltaIndex* where `np.sum(index)` incorrectly returned

a zerodimensional object instead of a scalar (GH25282)

Fixed regression in IntervalDtype construction where passing an incorrect string with 'Interval' as a prefix could result in a RecursionError. (GH25338)

Fixed regression in creating a period-dtype array from a read-only NumPy array of period objects. (GH25403)

Fixed regression in *Categorical*, where constructing it from a categorical Series and an explicit categories= that differed from that in the Series created an invalid object which could trigger segfaults. (GH25318)

Fixed regression in *to_timedelta()* losing precision when converting floating data to Timedelta data (GH25077).

Fixed pip installing from source into an environment without NumPy

(GH25193) Fixed regression in *DataFrame.replace()* where large strings of numbers would be coerced into int64, causing an OverflowError (GH25616)

Fixed regression in *factorize()* when passing a custom na_sentinel value with sort=True (GH25409).

Fixed regression in *DataFrame.to_csv()* writing duplicate line endings with gzip compress (GH25311)

Bug fixes

I/O

Better handling of terminal printing when the terminal dimensions are not known (GH25080)

Bug in reading a HDF5 table-format DataFrame created in Python 2, in Python 3 (GH24925)

Bug in reading a JSON with orient='table' generated by *DataFrame.to_json()* with index=False (GH25170)

Bug where float indexes could have misaligned values when printing (GH25061) Categorical

Bug where calling *Series.replace()* on categorical data could return a Series with incorrect dimensions (GH24971)

-

Reshaping

Bug in `transform()` where applying a function to a timezone aware column would return a timezone naive result (GH24198)

Bug in `DataFrame.join()` when joining on a timezone aware `DatetimeIndex` (GH23931) Visualization

Bug in `Series.plot()` where a secondary y axis could not be set to log scale (GH25545) Other

Bug in `Series.is_unique()` where single occurrences of NaN were not considered unique (GH25180)

Bug in `merge()` when merging an empty DataFrame with an Int64 column or a nonempty DataFrame with an Int64 column that is all NaN (GH25183)

Bug in `IntervalTree` where a `RecursionError` occurs upon construction due to an overflow when adding endpoints, which also causes `IntervalIndex` to crash during indexing operations (GH25485)

Bug in `Series.size` raising for some extension-array-backed Series, rather than returning the size (GH25580)

Bug in resampling raising for nullable integer-dtype columns (GH25580) 5.5.

Version 0.24

Contributors

A total of 25 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Albert Villanova del Moral

Arno Veenstra +

chris-b1

Devin Petersohn +

EternalLearner42 +

Flavien Lambert +

gfyoun

Gioia Ballin

jbrockmendel

Jeff Reback

Jeremy Schendel

Johan von Forstner +
Joris Van den Bossche
Josh
Justin Zheng
Kendall Masse
Matthew Roeschke Max Bolingbroke + rbenes +
Sterling Paramore + Tao He +
Thomas A Caswell Tom Augspurger Vibhu Agarwal + William Ayd
Zach Angell
5.5.2

Warning: The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See [Dropping Python 2.7](#) for more.

These are the changes in 0.24.1. See *Release notes* for a full changelog including other versions of . See *What's new in 0.24.0 (January 25, 2019)* for the 0.24.0 changelog.

API changes

Changing the sort parameter for Index set operations

The default sort value for *Index.union()* has changed from True to None (GH24959). The default *behavior*, however, remains the same: the result is sorted, unless

self and other are identical

self or other is empty

self or other contain values that can not be compared (a RuntimeWarning is raised). This change will allow sort=True to mean “always sort” in a future release.

The same change applies to *Index.difference()* and *Index.symmetric_difference()*, which would not sort the result when the values could not be compared.

The sort option for *Index.intersection()* has changed in three ways.

The default has changed from True to False, to restore the 0.23.4 and earlier behavior of not sorting by default.

The behavior of `sort=True` can now be obtained with `sort=None`. This will sort the result only if the values in `self` and `other` are not identical. The value `sort=True` is no longer allowed. A future version of will properly support `sort=True` meaning “always sort”.

Fixed regressions

Fixed regression in `DataFrame.to_dict()` with `records` orient raising an `AttributeError` when the `DataFrame` contained more than 255 columns, or wrongly converting column names that were not valid python identifiers (GH24939, GH24940).

Fixed regression in `read_sql()` when passing certain queries with MySQL/pymysql (GH24988).

Fixed regression in `Index.intersection` incorrectly sorting the values by default (GH24959).

Fixed regression in `merge()` when merging an empty `DataFrame` with multiple `timezoneaware` columns on one of the `timezone-aware` columns (GH25014).

5.5. Version 0.24.715

Fixed regression in `Series.rename_axis()` and `DataFrame.rename_axis()` where passing `None` failed to remove the axis name (GH25034)

Fixed regression in `to_timedelta()` with `box=False` incorrectly returning a `datetime64` object instead of a `timedelta64` object (GH24961)

Fixed regression where custom hashable types could not be used as column keys in `DataFrame`.

`set_index()` (GH24969)

Bug fixes

Reshaping

Bug in `DataFrame.groupby()` with `Grouper` when there is a time change (DST) and grouping frequency is `'1d'` (GH24972)

Visualization

Fixed the warning for implicitly registered matplotlib converters not showing. See *Restore Matplotlib datetime converter registration* for more (GH24963). Other

Fixed `AttributeError` when printing a `DataFrame`'s HTML repr after

accessing the IPython config object

(GH25036)

Contributors

A total of 7 people contributed patches to this release. People with a “+” by their names contributed a patch for the first time.

Alex Buchkovsky

Roman Yurchak

h-vetinari

jbrockmendel

Jeremy Schendel

Joris Van den Bossche

Tom Augspurger

Warning: The 0.24.x series of releases will be the last to support Python 2. Future feature releases will support Python 3 only. See [Dropping Python 2.7](#) for more details.

This is a major release from 0.23.4 and includes a number of API changes, new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

Optional Integer NA Support

New APIs for accessing the array backing a Series or Index

A new top-level method for creating arrays

Store Interval and Period data in a Series or DataFrame

Support for joining on two MultiIndexes

Check the *API Changes* and *deprecations* before updating.

These are the changes in 0.24.0. See *Release notes* for a full changelog including other versions of .

Enhancements

Optional integer NA support

has gained the ability to hold integer dtypes with missing values. This long requested feature is enabled through the use of *extension types*.

Note: IntegerArray is currently experimental. Its API or implementation may change without warning.

We can construct a Series with the specified dtype. The dtype string Int64 is a ExtensionDtype. Specifying a list or array using the traditional missing value marker of np.nan will infer to integer dtype. The display of the Series will also use the NaN to indicate missing values in string outputs. (GH20700, GH20747, GH22441, GH21789, GH22346)

```
In [1]: s = pd.Series([1, 2, np.nan], dtype='Int64')
```

```
In [2]: s Out[2]:
```

```
1
```

```
2
```

```
<NA>
```

```
Length: 3, dtype: Int64
```

```
Operations on these dtypes will propagate NaN as other operations. #  
arithmetic
```

```
In [3]: s + 1
```

```
Out[3]:
```

```
2
```

```
3
```

```
<NA>
```

```
Length: 3, dtype: Int64
```

```
# comparison In [4]: s == 1
```

```
Out[4]:
```

```
True
```

```
False
```

```
<NA>
```

```
Length: 3, dtype: boolean
```

```
# indexing
```

```
In [5]: s.iloc[1:3] Out[5]:
```

```
2
```

```
<NA>
```

```
Length: 2, dtype: Int64
```

```
# operate with other dtypes
```

```
In [6]: s + s.iloc[1:3].astype('Int8') Out[6]:
```

```
<NA>
```

4

<NA>

Length: 3, dtype: Int64

coerce when needed

In [7]: s + 0.01

Out[7]:

1.01

2.01

<NA>

Length: 3, dtype: Float64

These dtypes can operate as part of a DataFrame. **In [8]:** df =
pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab')}) **In [9]:** df

Out[9]:

A B C

1 1 a

2 1 a

<NA> 3 b

[3 rows x 3 columns]

In [10]: df.dtypes **Out[10]:**

Int64

int64

object

Length: 3, dtype: object

These dtypes can be merged, reshaped, and casted. **In [11]:**
pd.concat([df[['A']], df[['B', 'C']], axis=1).dtypes **Out[11]:**

Int64

int64

object

Length: 3, dtype: object

In [12]: df['A'].astype(float) **Out[12]:**

1.0

2.0

NaN

Name: A, Length: 3, dtype: float64

Reduction and groupby operations such as sum work.

In [13]: df.sum() **Out[13]:**

```
3
5
aab
Length: 3, dtype: object
In [14]: df.groupby('B').A.sum() Out[14]:
B
(continues on next page)
1 3
3 0
Name: A, Length: 2, dtype: Int64
```

Warning: The Integer NA support currently uses the capitalized dtype version, e.g. Int8 as compared to the traditional int8. This may be changed at a future date. See *Nullable integer data type* for more.

Accessing the values in a Series or Index

Series.array and *Index.array* have been added for extracting the array backing a Series or Index. (GH19954, GH23623)

```
In [15]: idx = pd.period_range('2000', periods=4)
In [16]: idx.array Out[16]:
<PeriodArray>
['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04']
Length: 4, dtype: period[D]
In [17]: pd.Series(idx).array Out[17]:
<PeriodArray>
['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04']
Length: 4, dtype: period[D]
```

Historically, this would have been done with `series.values`, but with `.values` it was unclear whether the returned value would be the actual array, some transformation of it, or one of custom arrays (like Categorical). For example, with *PeriodIndex*, `.values` generates a new ndarray of period objects each time.

```
In [18]: idx.values Out[18]:
array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)
In [19]: id(idx.values)
```

Out[19]: 140542227169424

In [20]: id(idx.values)

Out[20]: 140542173832656

If you need an actual NumPy array, use *Series.to_numpy()* or

Index.to_numpy(). **In [21]:** idx.to_numpy() **Out[21]:**

array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)

In [22]: pd.Series(idx).to_numpy()

Out[22]:

array([Period('2000-01-01', 'D'), Period('2000-01-02', 'D'),
Period('2000-01-03', 'D'), Period('2000-01-04', 'D')], dtype=object)

For Series and Indexes backed by normal NumPy arrays, *Series.array* will return a new *arrays.Array*, which is a thin (no-copy) wrapper around a `numpy.ndarray`. *Array* isn't especially useful on its own, but it does provide the same interface as any extension array defined in or by a third-party library.

In [23]: ser = pd.Series([1, 2, 3])

In [24]: ser.array **Out[24]:**

<Array>

[1, 2, 3]

Length: 3, dtype: int64

In [25]: ser.to_numpy()

Out[25]: array([1, 2, 3])

We haven't removed or deprecated *Series.values* or *DataFrame.values*, but we highly recommend and using *.array* or *.to_numpy()* instead.

See *Dtypes* and *Attributes and Underlying Data* for more.

.array: a new top-level method for creating arrays

A new top-level method *array()* has been added for creating 1-dimensional arrays (GH22860). This can be used to create any *extension array*, including extension arrays registered by 3rd party libraries. See the *dtypes docs* for more on extension arrays.

In [26]: pd.array([1, 2, np.nan], dtype='Int64') **Out[26]:**

<IntegerArray>

[1, 2, <NA>]

Length: 3, dtype: Int64

In [27]: pd.array(['a', 'b', 'c'], dtype='category') **Out[27]:**

['a', 'b', 'c']

Categories (3, object): ['a', 'b', 'c']

Passing data for which there isn't dedicated extension type (e.g. float, integer, etc.) will return a new *arrays. Array*, which is just a thin (no-copy) wrapper around a `numpy.ndarray` that satisfies the extension array interface.

In [28]: pd.array([1, 2, 3]) **Out[28]:**

<IntegerArray>

[1, 2, 3]

Length: 3, dtype: Int64

On their own, a *Array* isn't a very useful object. But if you need write low-level code that works generically for any *ExtensionArray*, *Array* satisfies that need.

Notice that by default, if no dtype is specified, the dtype of the returned array is inferred from the data. In particular, note that the first example of [1, 2, np.nan] would have returned a floating-point array, since NaN is a float.

In [29]: pd.array([1, 2, np.nan]) **Out[29]:**

<IntegerArray>

[1, 2, <NA>]

Length: 3, dtype: Int64

Storing Interval and Period data in Series and DataFrame

Interval and *Period* data may now be stored in a *Series* or *DataFrame*, in addition to an *IntervalIndex* and *PeriodIndex* like previously (GH19453, GH22862).

In [30]: ser = pd.Series(pd.interval_range(0, 5))

In [31]: ser **Out[31]:**

(0, 1]

(1, 2]

(2, 3]

(3, 4]

(4, 5]

Length: 5, dtype: interval

In [32]: ser.dtype

Out[32]: interval[int64]

For periods:

In [33]: pser = pd.Series(pd.period_range("2000", freq="D", periods=5)) **In**

[34]: pser **Out[34]:**

2000-01-01

2000-01-02

2000-01-03

2000-01-04

2000-01-05

Length: 5, dtype: period[D]

In [35]: pser.dtype **Out[35]:** period[D]

Previously, these would be cast to a NumPy array with object dtype. In general, this should result in better performance when storing an array of intervals or periods in a *Series* or column of a *DataFrame*.

Use *Series.array* to extract the underlying array of intervals or periods from the Series: **In [36]:** ser.array **Out[36]:**

<IntervalArray>

[(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]

Length: 5, closed: right, dtype: interval[int64]

In [37]: pser.array

5.5. Version 0.24

(continued from previous page)

Out[37]:

<PeriodArray>

['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04', '2000-01-05']

Length: 5, dtype: period[D]

These return an instance of *arrays.IntervalArray* or *arrays.PeriodArray*, the new extension arrays that back interval and period data.

Warning: For backwards compatibility, *Series.values* continues to return a

NumPy array of objects for Interval and Period data. We recommend using *Series.array* when you need the array of data stored in the Series, and *Series.to_numpy()* when you know you need a NumPy array.

See *Dtypes* and *Attributes and Underlying Data* for more.

Joining with two multi-indexes

DataFrame.merge() and *DataFrame.join()* can now be used to join multi-indexed Dataframe instances on the overlapping index levels (GH6360)

See the *Merge, join, and concatenate* documentation section. **In [38]:**
index_left = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'), ...: ('K1', 'X2')], ...: names=['key', 'X'])

....:

In [39]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],

....: 'B': ['B0', 'B1', 'B2']}, index=index_left)

....:

In [40]: index_right = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),

....: ('K2', 'Y2'), ('K2', 'Y3')],

....: names=['key', 'Y'])

....:

In [41]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],

....: 'D': ['D0', 'D1', 'D2', 'D3']}, index=index_right):

In [42]: left.join(right) **Out[42]:**

A B C D

key X Y

K0 X0 Y0 A0 B0 C0 D0

X1 Y0 A1 B1 C0 D0

K1 X2 Y1 A2 B2 C1 D1

[3 rows x 4 columns]

For earlier versions this can be done using the following.

In [43]: pd.merge(left.reset_index(), right.reset_index(),

....: on=['key'], how='inner').set_index(['key', 'X', 'Y']): (continues on next page)

(continued from previous page)

Out[43]:

A B C D

key X Y

K0 X0 Y0 A0 B0 C0 D0

X1 Y0 A1 B1 C0 D0

K1 X2 Y1 A2 B2 C1 D1

[3 rows x 4 columns]

Function read_html enhancements

read_html() previously ignored colspan and rowspan attributes. Now it understands them, treating them as sequences of cells with the same value. (GH17054)

In [44]: result = pd.read_html("""

....: <table>

....: <thead>

....: <tr>

....: <th>A</th><th>B</th><th>C</th>

....: </tr>

....: </thead>

....: <tbody>

....: <tr>

....: <td colspan="2">1</td><td>2</td>

....: </tr>

....: </tbody>: </table>"""):

Previous behavior:

In [13]: result **Out [13]:**

[A B C

0 1 2 NaN]

New behavior:

In [45]: result **Out[45]:**

[A B C

0 1 1 2

[1 rows x 3 columns]]

5.5. Version 0.24

New Styler.pipe() method

The *Styler* class has gained a *pipe()* method. This provides a convenient way to apply users' predefined styling functions, and can help reduce "boilerplate" when using DataFrame styling functionality repeatedly within a notebook. (GH23229)

```
In [46]: df = pd.DataFrame({'N': [1250, 1500, 1750], 'X': [0.25, 0.35, 0.50]})
```

```
In [47]: def format_and_align(styler):
```

```
....: return (styler.format({'N': '{:,}', 'X': '{:.1%}'}) ....: .set_properties(**
{'text-align': 'right'}))
```

```
....:
```

```
In [48]: df.style.pipe(format_and_align).set_caption('Summary of results.')
```

```
Out[48]: <.io.formats.style.Styler at 0x7fd28d6fb100>
```

Similar methods already exist for other classes in , including *DataFrame.pipe()*, *GroupBy.pipe()*, and *Resampler.pipe()*.

Renaming names in a MultiIndex

DataFrame.rename_axis() now supports index and columns arguments and *Series.rename_axis()* supports index argument (GH19978).

This change allows a dictionary to be passed so that some of the names of a MultiIndex can be changed.

Example:

```
In [49]: mi = pd.MultiIndex.from_product([list('AB'), list('CD'), list('EF')],
```

```
....: names=['AB', 'CD', 'EF'])
```

```
....:
```

```
In [50]: df = pd.DataFrame(list(range(len(mi))), index=mi, columns=['N']) In
```

```
[51]: df Out[51]:
```

N

AB CD EF

C E 0

F 1

E 2

3

C E 4

F 5

E 6

7

[8 rows x 1 columns]

In [52]: df.rename_axis(index={'CD': 'New'}) **Out[52]:**

N

AB New EF

A C E 0

F 1

D E 2

(continues on next page)

(continued from previous page)

F 3

B C E 4

F 5

D E 6

F 7

[8 rows x 1 columns]

See the *Advanced documentation on renaming* for more details.

Other enhancements

merge() now directly allows merge between objects of type DataFrame and named Series, without the need to convert the Series object into a DataFrame beforehand (GH21220)

ExcelWriter now accepts mode as a keyword argument, enabling append to existing workbooks when using the openpyxl engine (GH3441)

FrozenList has gained the *.union()* and *.difference()* methods. This functionality greatly simplifies groupby's that rely on explicitly excluding certain columns. See *Splitting an object into groups* for more information (GH15475, GH15506).

DataFrame.to_parquet() now accepts index as an argument, allowing the user to override the engine's default behavior to include or omit the dataframe's indexes from the resulting Parquet file. (GH20768)

read_feather() now accepts columns as an argument, allowing the user to specify which columns should be read. (GH24025)

DataFrame.corr() and *Series.corr()* now accept a callable for generic calculation methods of correlation, e.g. histogram intersection (GH22684)

DataFrame.to_string() now accepts decimal as an argument, allowing the user to specify which decimal separator should be used in the output. (GH23614)

DataFrame.to_html() now accepts *render_links* as an argument, allowing the user to generate HTML with links to any URLs that appear in the DataFrame. See the *section on writing HTML* in the IO docs for example usage. (GH2679)

.read_csv() now supports extension types as an argument to *dtype*, allowing the user to use extension types when reading CSVs. (GH23228)

The *shift()* method now accepts *fill_value* as an argument, allowing the user to specify a value which will be used instead of NA/NaT in the empty periods. (GH15486) *to_datetime()* now supports the %Z and %z directive when passed into *format* (GH13486)

Series.mode() and *DataFrame.mode()* now support the *dropna* parameter which can be used to specify whether NaN/NaT values should be considered (GH17534) *DataFrame.to_csv()* and *Series.to_csv()* now support the *compression* keyword when a file handle is passed. (GH21227)

Index.droplevel() is now implemented also for flat indexes, for compatibility with *MultiIndex* (GH21115)

Series.droplevel() and *DataFrame.droplevel()* are now implemented (GH20342) Added support for reading from/writing to Google Cloud Storage via the *gcsfs* library (GH19454, GH23094)

5.5. Version 0.24

DataFrame.to_gbq() and *read_gbq()* signature and documentation updated to reflect changes from the -gbq library version 0.8.0. Adds a *credentials* argument, which enables the use of any kind of google-auth credentials. (GH21627, GH22557, GH23662)

New method *HDFStore.walk()* will recursively walk the group hierarchy of

an HDF5 file (GH10932)

read_html() copies cell data across colspan and rowspan, and it treats all-table rows as headers if header kwarg is not given and there is no thead (GH17054)

Series.nlargest() , *Series.nsmallest()*, *DataFrame.nlargest()*, and *DataFrame.nsmallest()* now accept the value "all" for the keep argument. This keeps all ties for the nth largest/smallest value (GH16818)

IntervalIndex has gained the *set_closed()* method to change the existing closed value (GH21670)

to_csv() , *to_csv()*, *to_json()*, and *to_json()* now support compression='infer' to infer compression based on filename extension (GH15008). The default compression for *to_csv*, *to_json*, and *to_pickle* methods has been updated to 'infer' (GH22004).

DataFrame.to_sql() now supports writing TIMESTAMP WITH TIME ZONE types for supported databases. For databases that don't support timezones, datetime data will be stored as timezone unaware local timestamps. See the *Datetime data types* for implications (GH9086).

to_timedelta() now supports iso-formatted timedelta strings (GH21877)
Series and *DataFrame* now support Iterable objects in the constructor (GH2193)

DatetimeIndex has gained the *DatetimeIndex.timetz* attribute. This returns the local time with timezone information. (GH21358)

round() , *ceil()*, and *floor()* for *DatetimeIndex* and *Timestamp* now support an ambiguous argument for handling datetimes that are rounded to ambiguous times (GH18946) and a nonexistent argument for handling datetimes that are rounded to nonexistent times. See *Nonexistent times when localizing* (GH22647)

The result of *resample()* is now iterable similar to *groupby()* (GH15314).
Series.resample() and *DataFrame.resample()* have gained the *.core.resample.Resampler.quantile()* (GH15023).

DataFrame.resample() and *Series.resample()* with a *PeriodIndex* will now respect the base argument in the same fashion as with a *DatetimeIndex*. (GH23882) *.api.types.is_list_like()* has gained a keyword *allow_sets* which is True by default; if False, all instances of set will not be considered “list-like” anymore (GH23061) *Index.to_frame()* now supports overriding column name(s) (GH22580). *Categorical.from_codes()* now can take a dtype parameter as an alternative to passing categories and ordered (GH24398).

New attribute *__git_version__* will return git commit sha of current build (GH21295). Compatibility with Matplotlib 3.0 (GH22790).

Added *Interval.overlaps()*, *arrays.IntervalArray.overlaps()*, and *IntervalIndex.overlaps()* for determining overlaps between interval-like objects (GH21998)

read_fwf() now accepts keyword *infer_nrows* (GH15138).

to_parquet() now supports writing a DataFrame as a directory of parquet files partitioned by a subset of the columns when *engine = 'pyarrow'* (GH23283)

Timestamp.tz_localize(), *DatetimeIndex.tz_localize()*, and have gained the nonexistent argument for alternative nonexistent times. See *Nonexistent times when localizing* (GH8917, GH24466)

Index.difference() , *Index.intersection()*, *Index.union()*, and *Index.symmetric_difference()* now have an optional sort parameter to control whether the results should be sorted if possible (GH17839, GH24471)

read_excel() now accepts *usecols* as a list of column names or callable (GH18273) *MultiIndex.to_flat_index()* has been added to flatten multiple levels into a single-level *Index* object.

DataFrame.to_stata() and *.io.stata.StataWriter*117 can write mixed string columns to Stata strl format (GH23633)

DataFrame.between_time() and *DataFrame.at_time()* have gained the axis parameter (GH8839)

DataFrame.to_records() now accepts *index_dtypes* and *column_dtypes* parameters to allow different data types in stored column and index records (GH18146) *IntervalIndex* has gained the *is_overlapping* attribute to indicate if the *IntervalIndex* contains any overlapping intervals (GH23309)

Series.tz_localize() handling of

`DataFrame.to_sql()` has gained the `method` argument to control SQL insertion clause. See the *insertion method* section in the documentation. (GH8953)

`DataFrame.corrwith()` now supports Spearman's rank correlation, Kendall's tau as well as callable correlation methods. (GH21925)

`DataFrame.to_json()`, `DataFrame.to_csv()`, `DataFrame.to_pickle()`, and other export methods now support tilde(~) in path argument. (GH23473)

Backwards incompatible API changes 0.24.0 includes a number of API breaking changes.

Increased minimum versions for dependencies

We have updated our minimum supported versions of dependencies (GH21242, GH18742, GH23774, GH24767). If installed, we now require:

Package Minimum Required Version

numpy 1.12.0 X bottleneck 1.2.0 fastparquet 0.2.1 matplotlib 2.0.0 numexpr 2.6.1

-gbq 0.8.0 pyarrow 0.9.0 pytables 3.4.2 scipy 0.18.1 xlrd 1.0.0 pytest (dev)

5.5. Version 0.24

3.6

Additionally we no longer depend on feather-format for feather based storage and replaced it with references to pyarrow (GH21639 and GH23053).

os.linesep is used for line_terminator of DataFrame.to_csv

`DataFrame.to_csv()` now uses `os.linesep()` rather than `'\n'` for the default line terminator (GH20353). This change only affects when running on Windows, where `'\r\n'` was used for line terminator even when `'\n'` was passed in `line_terminator`.

Previous behavior on Windows:

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
```

```
...: "string_with_crlf": ["a\r\nbc"]})
```

```
In [2]: # When passing file PATH to to_csv,
```

```
...: # line_terminator does not work, and csv is saved with '\r\n'.
```

```
...: # Also, this converts all '\n's in the data to '\r\n'.
```

```
...: data.to_csv("test.csv", index=False, line_terminator='\n')
```

```
In [3]: with open("test.csv", mode='rb') as f:
```

```
...: print(f.read())
```

```
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\r\nbc","a\r\nbc"\r\n'
```

In [4]: *# When passing file OBJECT with newline option to ...: # to_csv, line_terminator works.*

```
...: with open("test2.csv", mode='w', newline='\n') as f:
```

```
...: data.to_csv(f, index=False, line_terminator='\n')
```

```
In [5]: with open("test2.csv", mode='rb') as f:
```

```
...: print(f.read())
```

```
Out[5]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'
```

New behavior on Windows:

Passing line_terminator explicitly, set the line terminator to that character.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
```

```
...: "string_with_crlf": ["a\r\nbc"]})
```

```
In [2]: data.to_csv("test.csv", index=False, line_terminator='\n')
```

```
In [3]: with open("test.csv", mode='rb') as f:
```

```
...: print(f.read())
```

```
Out[3]: b'string_with_lf,string_with_crlf\n"a\nbc","a\r\nbc"\n'
```

On Windows, the value of os.linesep is '\r\n', so if line_terminator is not set, '\r\n' is used for line terminator.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"], ...:
```

```
"string_with_crlf": ["a\r\nbc"]})
```

```
In [2]: data.to_csv("test.csv", index=False)
```

```
In [3]: with open("test.csv", mode='rb') as f:
```

```
...: print(f.read())
```

```
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\nbc","a\r\nbc"\r\n'
```

For file objects, specifying newline is not sufficient to set the line terminator. You must pass in the line_terminator explicitly, even in this case.

```
In [1]: data = pd.DataFrame({"string_with_lf": ["a\nbc"],
```

```
...: "string_with_crlf": ["a\r\nbc"]})
```

```
In [2]: with open("test2.csv", mode='w', newline='\n') as f:
```

```
...: data.to_csv(f, index=False)
```

```
In [3]: with open("test2.csv", mode='rb') as f:
```

```
...: print(f.read())
```

```
Out[3]: b'string_with_lf,string_with_crlf\r\n"a\nbc","a\r\nbc"\r\n'
```

Proper handling of np.NaN in a string data-typed column with the Python engine

There was bug in `read_excel()` and `read_csv()` with the Python engine, where missing values turned to 'nan' with `dtype=str` and `na_filter=True`. Now, these missing values are converted to the string missing indicator, `np.nan`. (GH20377)

Previous behavior :

```
In [5]: data = 'a,b,c\n1,,3\n4,5,6'
```

```
In [6]: df = pd.read_csv(StringIO(data), engine='python', dtype=str,  
na_filter=True) In [7]: df.loc[0, 'b'] Out[7]:  
'nan'
```

New behavior:

```
In [53]: data = 'a,b,c\n1,,3\n4,5,6'
```

```
In [54]: df = pd.read_csv(StringIO(data), engine='python', dtype=str,  
na_filter=True) In [55]: df.loc[0, 'b']
```

```
Out[55]: nan
```

Notice how we now instead output `np.nan` itself instead of a stringified form of it. **Parsing datetime strings with timezone offsets**

Previously, parsing datetime strings with UTC offsets with `to_datetime()` or `DatetimeIndex` would automatically convert the datetime to UTC without timezone localization. This is inconsistent from parsing the same datetime string with `Timestamp` which would preserve the UTC offset in the `tz` attribute. Now, `to_datetime()` preserves the UTC offset in the `tz` attribute when all the datetime strings have the same UTC offset (GH17697, GH11736, GH22457) *Previous behavior:*

```
In [2]: pd.to_datetime("2015-11-18 15:30:00+05:30")
```

```
Out[2]: Timestamp('2015-11-18 10:00:00')
```

```
In [3]: pd.Timestamp("2015-11-18 15:30:00+05:30")
```

```
Out[3]: Timestamp('2015-11-18 15:30:00+0530', tz='pytz.FixedOffset(330))
```

Different UTC offsets would automatically convert the datetimes to UTC (without a , → UTC timezone) (continues on next page) (continued from previous page) **In [4]:** pd.to_datetime(["2015-11-18 15:30:00+05:30", "2015-

```
11-18 16:30:00+06:30"]) Out[4]: DatetimeIndex(['2015-11-18 10:00:00',  
'2015-11-18 10:00:00'], dtype= , → 'datetime64[ns]', freq=None)
```

New behavior:

```
In [56]: pd.to_datetime("2015-11-18 15:30:00+05:30")
```

```
Out[56]: Timestamp('2015-11-18 15:30:00+0530',  
tz='pytz.FixedOffset(330)') In [57]: pd.Timestamp("2015-11-18  
15:30:00+05:30")
```

```
Out[57]: Timestamp('2015-11-18 15:30:00+0530',  
tz='pytz.FixedOffset(330)') Parsing datetime strings with the same UTC  
offset will preserve the UTC offset in the tz In [58]: pd.to_datetime(["2015-  
11-18 15:30:00+05:30"] * 2)
```

```
Out[58]: DatetimeIndex(['2015-11-18 15:30:00+05:30', '2015-11-18  
15:30:00+05:30'], , → dtype='datetime64[ns, pytz.FixedOffset(330)]',  
freq=None)
```

Parsing datetime strings with different UTC offsets will now create an Index of datetime.datetime objects with different UTC offsets

```
In [59]: idx = pd.to_datetime(["2015-11-18 15:30:00+05:30",  
....: "2015-11-18 16:30:00+06:30"])  
....:
```

```
In [60]: idx
```

```
Out[60]: Index([2015-11-18 15:30:00+05:30, 2015-11-18 16:30:00+06:30],  
dtype='object')
```

```
In [61]: idx[0]
```

```
Out[61]: datetime.datetime(2015, 11, 18, 15, 30, tzinfo=tzoffset(None,  
19800))
```

```
In [62]: idx[1]
```

```
Out[62]: datetime.datetime(2015, 11, 18, 16, 30, tzinfo=tzoffset(None,  
23400))
```

Passing utc=True will mimic the previous behavior but will correctly indicate that the dates have been converted to UTC

```
In [63]: pd.to_datetime(["2015-11-18 15:30:00+05:30",
```

```
....: "2015-11-18 16:30:00+06:30"], utc=True)
```

....:

Out[63]: DatetimeIndex(['2015-11-18 10:00:00+00:00', '2015-11-18 10:00:00+00:00'], , _dtype='datetime64[ns, UTC]', freq=None)

Parsing mixed-timezones with read_csv()

read_csv() no longer silently converts mixed-timezone columns to UTC (GH24987). *Previous behavior*

```
>>> import io >>> content = """"\
```

```
... a
```

```
... 2000-01-01T00:00:00+05:00
```

```
... 2000-01-01T00:00:00+06:00""""
```

```
>>> df = pd.read_csv(io.StringIO(content), parse_dates=['a'])
```

(continues on next page)

(continued from previous page)

```
>>> df.a
```

```
1999-12-31 19:00:00
```

```
1999-12-31 18:00:00
```

```
Name: a, dtype: datetime64[ns]
```

New behavior

In [64]: import io

In [65]: content = """"\

....: a

....: 2000-01-01T00:00:00+05:00: 2000-01-01T00:00:00+06:00"""": **In**

[66]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'])

In [67]: df.a **Out[67]:**

```
2000-01-01 00:00:00+05:00
```

```
2000-01-01 00:00:00+06:00
```

```
Name: a, Length: 2, dtype: object
```

As can be seen, the dtype is object; each value in the column is a string. To convert the strings to an array of datetimes, the `date_parser` argument

In [68]: df = pd.read_csv(io.StringIO(content), parse_dates=['a'],

....: `date_parser=`**lambda** col: pd.to_datetime(col, utc=**True**))

....:

In [69]: df.a **Out[69]:**

```
1999-12-31 19:00:00+00:00
```

1999-12-31 18:00:00+00:00

Name: a, Length: 2, dtype: datetime64[ns, UTC]

See *Parsing datetime strings with timezone offsets* for more.

Time values in `dt.end_time` and `to_timestamp(how='end')`

The time values in *Period* and *PeriodIndex* objects are now set to

'23:59:59.999999999' when calling

Series.dt.end_time, *Period.end_time*, *PeriodIndex.end_time*,

Period.to_timestamp() with `how='end'`, or *PeriodIndex.to_timestamp()* with `how='end'` (GH17157)

Previous behavior :

In [2]: p = pd.Period('2017-01-01', 'D')

In [3]: pi = pd.PeriodIndex([p])

In [4]: pd.Series(pi).dt.end_time[0]

Out[4]: Timestamp(2017-01-01 00:00:00)

In [5]: p.end_time **Out[5]:** Timestamp(2017-01-01 23:59:59.999999999)

New behavior:

Calling *Series.dt.end_time* will now result in a time of '23:59:59.999999999' as is the case with *Period.end_time*, for example

In [70]: p = pd.Period('2017-01-01', 'D')

In [71]: pi = pd.PeriodIndex([p])

In [72]: pd.Series(pi).dt.end_time[0]

Out[72]: Timestamp('2017-01-01 23:59:59.999999999')

In [73]: p.end_time **Out[73]:** Timestamp('2017-01-01 23:59:59.999999999')

Series.unique for timezone-aware data

The return type of *Series.unique()* for datetime with timezone values has changed from an `numpy.ndarray` of *Timestamp* objects to a *arrays.DatetimeArray* (GH24024).

In [74]: ser = pd.Series([pd.Timestamp('2000', tz='UTC'),

....: pd.Timestamp('2000', tz='UTC')])

....:

Previous behavior:

In [3]: ser.unique() **Out[3]:** array([Timestamp('2000-01-01 00:00:00+0000', tz='UTC')], dtype=object)

New behavior :

In [75]: ser.unique() **Out[75]:**

<DatetimeArray>

['2000-01-01 00:00:00+00:00']

Length: 1, dtype: datetime64[ns, UTC]

Sparse data structure refactor

SparseArray, the array backing SparseSeries and the columns in a SparseDataFrame, is now an extension array (GH21978, GH19056, GH22835). To conform to this interface and for consistency with the rest of , some API breaking changes were made:

SparseArray is no longer a subclass of numpy.ndarray. To convert a SparseArray to a NumPy array, use numpy.asarray().

SparseArray.dtype and SparseSeries.dtype are now instances of *SparseDtype*, rather than np.dtype. Access the underlying dtype with SparseDtype.subtype.

numpy.asarray(sparse_array) now returns a dense array with all the values, not just the non-fill-value values (GH14167)

SparseArray.take now matches the API of *.api.extensions.ExtensionArray.take()* (GH19506):

The default value of allow_fill has changed from False to True.

The out and mode parameters are now longer accepted (previously, this raised if they were specified).

Passing a scalar for indices is no longer allowed.

The result of *concat()* with a mix of sparse and dense Series is a Series with sparse values, rather than a SparseSeries.

SparseDataFrame.combine and DataFrame.combine_first no longer supports combining a sparse column with a dense column while preserving the sparse subtype. The result will be an object-dtype SparseArray.

Setting SparseArray.fill_value to a fill value with a different dtype is now allowed. DataFrame[column] is now a *Series* with sparse values, rather than a SparseSeries, when slicing a single column with sparse values (GH23559).

The result of `Series.where()` is now a Series with sparse values, like with other extension arrays (GH24077)

Some new warnings are issued for operations that require or are likely to materialize a large dense array:

A `errors.PerformanceWarning` is issued when using `fillna` with a method, as a dense array is constructed to create the filled array. Filling with a value is the efficient way to fill a sparse array.

A `errors.PerformanceWarning` is now issued when concatenating sparse Series with differing fill values. The fill value from the first sparse array continues to be used. In addition to these API breaking changes, many *Performance Improvements and Bug Fixes have been made*.

Finally, a `Series.sparse` accessor was added to provide sparse-specific methods like `Series.sparse.from_coo()`.

```
In [76]: s = pd.Series([0, 0, 1, 1, 1], dtype='Sparse[int]')
```

```
In [77]: s.sparse.density Out[77]: 0.6
```

`get_dummies()` always returns a DataFrame

Previously, when `sparse=True` was passed to `get_dummies()`, the return value could be either a `DataFrame` or a `SparseDataFrame`, depending on whether all or a just a subset of the columns were dummy-encoded. Now, a `DataFrame` is always returned (GH24284).

Previous behavior

The first `get_dummies()` returns a `DataFrame` because the column A is not dummy encoded. When just ["B", "C"] are passed to `get_dummies`, then all the columns are dummy-encoded, and a `SparseDataFrame` was returned.

```
In [2]: df = pd.DataFrame({"A": [1, 2], "B": ['a', 'b'], "C": ['a', 'a']})
```

```
In [3]: type(pd.get_dummies(df, sparse=True))
```

```
Out[3]: .core.frame.DataFrame
```

```
In [4]: type(pd.get_dummies(df[['B', 'C']], sparse=True))
```

```
Out[4]: .core.sparse.frame.SparseDataFrame
```

New behavior

Now, the return type is consistently a `DataFrame`.

```
In [78]: type(pd.get_dummies(df, sparse=True))
```


Out[78]: .core.frame.DataFrame

In [79]: type(pd.get_dummies(df[['B', 'C']], sparse=True))

Out[79]: .core.frame.DataFrame

Note: There's no difference in memory usage between a SparseDataFrame and a *DataFrame* with sparse values. The memory usage will be the same as in the previous version of .

Raise ValueError in DataFrame.to_dict(orient='index')

Bug in *DataFrame.to_dict()* raises ValueError when used with orient='index' and a nonunique index instead of losing data (GH22801)

In [80]: df = pd.DataFrame({'a': [1, 2], 'b': [0.5, 0.75]}, index=['A', 'A'])

In [81]: df **Out[81]:**

a b

A 1 0.50

A 2 0.75

[2 rows x 2 columns]

In [82]: df.to_dict(orient='index')

ValueError Traceback (most recent call last)

<ipython-input-82-f5309a7c6adb> in <module>

----> 1 df.to_dict(orient='index')

///core/frame.py in to_dict(self, orient, into)

elif orient == "index":

if not self.index.is_unique:

-> 1607 raise ValueError("DataFrame index must be unique for orient=, → 'index'.")

return into_c(

(t[0], dict(zip(self.columns, t[1:])))

ValueError: DataFrame index must be unique for orient='index'.

Tick DateOffset normalize restrictions

Creating a Tick object (Day, Hour, Minute, Second, Milli, Micro, Nano) with normalize=True is no longer supported. This prevents unexpected behavior where addition could fail to be monotone or associative. (GH21427)

Previous behavior :

```

In [2]: ts = pd.Timestamp('2018-06-11 18:01:14')
In [3]: ts
Out[3]: Timestamp('2018-06-11 18:01:14')
(continues on next page)
In [4]: tic = pd.offsets.Hour(n=2, normalize=True) ...:
In [5]: tic
Out[5]: <2 * Hours>
In [6]: ts + tic
Out[6]: Timestamp('2018-06-11 00:00:00')
In [7]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[7]: False
New behavior:
In [83]: ts = pd.Timestamp('2018-06-11 18:01:14')
In [84]: tic = pd.offsets.Hour(n=2)
In [85]: ts + tic + tic + tic == ts + (tic + tic + tic)
Out[85]: True
Period subtraction

```

Subtraction of a Period from another Period will give a DateOffset. instead of an integer (GH21314)

Previous behavior :

```

In [2]: june = pd.Period('June 2018') In [3]: april = pd.Period('April 2018')
In [4]: june - april
Out [4]: 2

```

New behavior:

```

In [86]: june = pd.Period('June 2018') In [87]: april = pd.Period('April 2018')
In [88]: june - april
Out[88]: <2 * MonthEnds>

```

Similarly, subtraction of a Period from a PeriodIndex will now return an Index of DateOffset objects instead of an Int64Index

Previous behavior :

```

In [2]: pi = pd.period_range('June 2018', freq='M', periods=3)
In [3]: pi - pi[0] Out[3]: Int64Index([0, 1, 2], dtype='int64')

```

New behavior :

In [89]: pi = pd.period_range('June 2018', freq='M', periods=3)

In [90]: pi - pi[0] **Out[90]:** Index([<0 * MonthEnds>, <MonthEnd>, <2 * MonthEnds>], dtype='object')

Addition/subtraction of NaN from DataFrame

Adding or subtracting NaN from a *DataFrame* column with `timedelta64[ns]` dtype will now raise a `TypeError` instead of returning all-`NaT`. This is for compatibility with `TimedeltaIndex` and `Series` behavior

(GH22163)

In [91]: df = pd.DataFrame([pd.Timedelta(days=1)])

In [92]: df **Out[92]:**

0

0 1 days

[1 rows x 1 columns]

Previous behavior:

In [4]: df = pd.DataFrame([pd.Timedelta(days=1)])

In [5]: df - np.nan

Out[5]:

0

0 NaT

New behavior:

In [2]: df - np.nan ...

`TypeError: unsupported operand type(s) for -: 'TimedeltaIndex' and 'float'`

DataFrame comparison operations broadcasting changes

Previously, the broadcasting behavior of *DataFrame* comparison operations (`==`, `!=`, `...`) was inconsistent with the behavior of arithmetic operations (`+`, `-`, `...`). The behavior of the comparison operations has been changed to match the arithmetic operations in these cases. (GH22880)

The affected cases are:

operating against a 2-dimensional `np.ndarray` with either 1 row or 1 column will now broadcast the same way a `np.ndarray` would (GH23000).

a list or tuple with length matching the number of rows in the *DataFrame* will now raise `ValueError` instead of operating column-by-column

(GH22880).

a list or tuple with length matching the number of columns in the *DataFrame* will now operate row-by-row instead of raising *ValueError* (GH22880).

In [93]: arr = np.arange(6).reshape(3, 2)

(continues on next page)

In [94]: df = pd.DataFrame(arr)

In [95]: df **Out[95]:**

0 1

0 1

2 3

4 5

[3 rows x 2 columns]

Previous behavior:

In [5]: df == arr[[0], :]

...: # comparison previously broadcast where arithmetic would raise **Out[5]:**

0 1

True True

False False

False False **In [6]:** df + arr[[0], :]

...

ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (1, 2)

In [7]: df == (1, 2)

...: # length matches number of columns;

...: # comparison previously raised where arithmetic would broadcast ...

ValueError: Invalid broadcasting comparison [(1, 2)] with block values **In**

[8]: df + (1, 2)

Out[8]:

0 1

1 3

3 5

5 7

In [9]: df == (1, 2, 3)

...: # length matches number of rows
...: # comparison previously broadcast where arithmetic would raise **Out[9]:**

0 1

False True

True False

False False **In [10]:** df + (1, 2, 3)

...

ValueError: Unable to coerce to Series, length must be 2: given 3

New behavior:

Comparison operations and arithmetic operations both broadcast. **In [96]:**
df == arr[[0], :] **Out[96]:**

0 1

True True

False False

False False

(continues on next page)

[3 rows x 2 columns]

In [97]: df + arr[[0], :]

Out[97]:

0 1

0 2

2 4

4 6

[3 rows x 2 columns]

Comparison operations and arithmetic operations both broadcast. **In [98]:**

df == (1, 2)

Out[98]:

0 1

False False

False False

False False

[3 rows x 2 columns]

In [99]: df + (1, 2)

Out[99]:

```

0 1
1 3
3 5
5 7
[3 rows x 2 columns]
# Comparison operations and arithmetic operations both raise ValueError. In
[6]: df == (1, 2, 3)
...
ValueError: Unable to coerce to Series, length must be 2: given 3 In [7]: df +
(1, 2, 3)

```

749 Chapter 5. Release notes

```

...
ValueError: Unable to coerce to Series, length must be 2: given 3
DataFrame arithmetic operations broadcasting changes

```

DataFrame arithmetic operations when operating with 2-dimensional `np.ndarray` objects now broadcast in the same way as `np.ndarray` broadcast. (GH23000)

```
In [100]: arr = np.arange(6).reshape(3, 2)
```

```
In [101]: df = pd.DataFrame(arr)
```

```
In [102]: df Out[102]:
```

```
0 1
```

```
0 1
```

```
2 3
```

```
(continues on next page)
```

```
2 4 5
```

```
[3 rows x 2 columns]
```

Previous behavior:

```
In [5]: df + arr[[0], :] # 1 row, 2 columns ...
```

```
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (1,
2) In [6]: df + arr[:, [1]] # 1 column, 3 rows ...
```

```
ValueError: Unable to coerce to DataFrame, shape must be (3, 2): given (3,
1) New behavior:
```

```
In [103]: df + arr[[0], :] # 1 row, 2 columns
```

Out[103]:

```
0 1
0 2
2 4
4 6
[3 rows x 2 columns]
```

In [104]: df + arr[:, [1]]

Out[104]:

```
0 1
1 2
5 6
9 10
[3 rows x 2 columns] # 1 column, 3 rows
```

Series and Index data-dtype incompatibilities

Series and Index constructors now raise when the data is incompatible with a passed dtype= (GH15832) *Previous behavior:*

In [4]: pd.Series([-1], dtype="uint64") **Out [4]:**

```
0 18446744073709551615 dtype: uint64
```

New behavior:

In [4]: pd.Series([-1], dtype="uint64") **Out [4]:**

...

OverflowError: Trying to coerce negative values to unsigned integers

Concatenation changes

Calling `.concat()` on a Categorical of ints with NA values now causes them to be processed as objects when concatenating with anything other than another Categorical of ints (GH19214)

In [105]: s = pd.Series([0, 1, np.nan]) **In [106]:** c = pd.Series([0, 1, np.nan], dtype="category")

Previous behavior

In [3]: pd.concat([s, c]) **Out[3]:**

```
0.0
```

```
1.0
NaN
0.0
1.0
NaN
dtype: float64
New behavior
In [107]: pd.concat([s, c]) Out[107]:
0.0
1.0
NaN
0.0
1.0
NaN
Length: 6, dtype: float64
Datetimelike API changes
```

For *DatetimeIndex* and *TimedeltaIndex* with non-Nonefreq attribute, addition or subtraction of integer-dtyped array or Index will return an object of the same class (GH19959)

DateOffset objects are now immutable. Attempting to alter one of these will now raise `AttributeError` (GH21341)

PeriodIndex subtraction of another *PeriodIndex* will now return an object-dtype *Index* of DateOffset objects instead of raising a `TypeError` (GH20049) *cut()* and *qcut()* now returns a *DatetimeIndex* or *TimedeltaIndex* bins when the input is datetime or timedelta dtype respectively and `retbins=True` (GH19891)

DatetimeIndex.to_period() and *Timestamp.to_period()* will issue a warning when timezone information will be lost (GH21333)

PeriodIndex.tz_convert() and *PeriodIndex.tz_localize()* have been removed (GH21781)

Other API changes

A newly constructed empty *DataFrame* with integer as the dtype will now only be cast to float64 if index is specified (GH22858)

Series.str.cat() will now raise if others is a set (GH23009)

Passing scalar values to *DatetimeIndex* or *TimedeltaIndex* will now raise

TypeError instead of ValueError (GH23539)

max_rows and max_cols parameters removed from HTMLFormatter since truncation is handled by DataFrameFormatter (GH23818) read_csv() will now raise a ValueError if a column with missing values is declared as having dtype bool (GH20591)

The column order of the resultant DataFrame from MultiIndex.to_frame() is now guaranteed to match the MultiIndex.names order. (GH22420)

Incorrectly passing a DatetimeIndex to MultiIndex.from_tuples(), rather than a sequence of tuples, now raises a TypeError rather than a ValueError (GH24024)

pd.offsets.generate_range() argument time_rule has been removed; use offset instead (GH24157)

In 0.23.x, would raise a ValueError on a merge of a numeric column (e.g. int dtyped column) and an object dtyped column (GH9780). We have re-enabled the ability to merge object and other dtypes; will still raise on a merge between a numeric and an object dtyped column that is composed only of strings (GH21681)

Accessing a level of a MultiIndex with a duplicate name (e.g. in get_level_values()) now raises a ValueError instead of a KeyError (GH21678).

Invalid construction of IntervalDtype will now always raise a TypeError rather than a ValueError if the subtype is invalid (GH21185)

753 Chapter 5. Release notes

Trying to reindex a DataFrame with a non unique MultiIndex now raises a ValueError instead of an Exception (GH21770)

Index subtraction will attempt to operate element-wise instead of raising TypeError (GH19369)

.io.formats.style.Styler supports a number-format property when using to_excel() (GH22015)

DataFrame.corr() and Series.corr() now raise a ValueError along with a helpful error message instead of a KeyError when supplied with an invalid

method (GH22298)

`shift()` will now always return a copy, instead of the previous behaviour of returning self when shifting by 0 (GH22397)

`DataFrame.set_index()` now gives a better (and less frequent) `KeyError`, raises a `ValueError` for incorrect types, and will not fail on duplicate column names with `drop=True`. (GH22484)

Slicing a single row of a `DataFrame` with multiple `ExtensionArrays` of the same type now preserves the dtype, rather than coercing to object (GH22784)

`DateOffset` attribute `_cacheable` and method `_should_cache` have been removed (GH23118)

`Series.searchsorted()`, when supplied a scalar value to search for, now returns a scalar instead of an array (GH23801).

`Categorical.searchsorted()`, when supplied a scalar value to search for, now returns a scalar instead of an array (GH23466).

5.5. Version 0.24

`Categorical.searchsorted()` now raises a `KeyError` rather than a `ValueError`, if a searched for key is not found in its categories (GH23466).

`Index.hasnans()` and `Series.hasnans()` now always return a python boolean. Previously, a python or a numpy boolean could be returned, depending on circumstances (GH23294).

The order of the arguments of `DataFrame.to_html()` and `DataFrame.to_string()` is rearranged to be consistent with each other. (GH23614)

`CategoricalIndex.reindex()` now raises a `ValueError` if the target index is non-unique and not equal to the current index. It previously only raised if the target index was not of a categorical dtype (GH23963).

`Series.to_list()` and `Index.to_list()` are now aliases of `Series.tolist` respectively `Index.tolist` (GH8826)

The result of `SparseSeries.unstack` is now a `DataFrame` with sparse values,

rather than a `SparseDataFrame` (GH24372).

`DatetimeIndex` and `TimedeltaIndex` no longer ignore the dtype precision. Passing a non-nanosecond resolution dtype will raise a `ValueError` (GH24753)

Extension type changes

Equality and hashability

now requires that extension dtypes be hashable (i.e. the respective `ExtensionDtype` objects; hashability is not a requirement for the values of the corresponding `ExtensionArray`). The base class implements a default `__eq__` and `__hash__`. If you have a parametrized dtype, you should update the `ExtensionDtype`. `_metadata` tuple to match the signature of your `__init__` method. See `.api.extensions.ExtensionDtype` for more (GH22476).

New and changed methods

`dropna()` has been added (GH21185)

`repeat()` has been added (GH24349)

The `ExtensionArray` constructor, `_from_sequence` now take the keyword arg `copy=False` (GH21185)

`.api.extensions.ExtensionArray.shift()` added as part of the basic

`ExtensionArray` interface (GH22387).

`searchsorted()` has been added (GH24350)

Support for reduction operations such as `sum`, `mean` via opt-in base class method override (GH22762)

`ExtensionArray.isna()` is allowed to return an `ExtensionArray` (GH22325).

Dtype changes

`ExtensionDtype` has gained the ability to instantiate from string dtypes, e.g. `decimal` would instantiate a registered `DecimalDtype`; furthermore the `ExtensionDtype` has gained the method `construct_array_type` (GH21185)

Added `ExtensionDtype._is_numeric` for controlling whether an extension dtype is considered numeric (GH22290).

Added `.api.types.register_extension_dtype()` to register an extension type with (GH22664)

Updated the `.type` attribute for `PeriodDtype`, `DatetimeTZDtype`, and `IntervalDtype` to be instances of the dtype (`Period`, `Timestamp`, and `Interval` respectively) (GH22938)

Operator support

A Series based on an `ExtensionArray` now supports arithmetic and comparison operators (GH19577). There are two approaches for providing operator support for an `ExtensionArray`:

Define each of the operators on your `ExtensionArray` subclass. Use an operator implementation from that depends on operators that are already defined on the underlying elements (scalars) of the `ExtensionArray`.

See the *ExtensionArray Operator Support* documentation section for details on both ways of adding operator support.

Other changes

A default repr for `.api.extensions.ExtensionArray` is now provided (GH23601).

`ExtensionArray._formatting_values()` is deprecated. Use `ExtensionArray._formatter` instead. (GH23601)

An `ExtensionArray` with a boolean dtype now works correctly as a boolean indexer. `.api.types.is_bool_dtype()` now properly considers them boolean (GH22326)

Bug fixes

Bug in `Series.get()` for Series using `ExtensionArray` and integer index (GH21257)

`shift()` now dispatches to `ExtensionArray.shift()` (GH22386) `Series.combine()` works correctly with `ExtensionArray` inside of `Series` (GH20825)

`Series.combine()` with scalar argument now works for any function type (GH21248)

`Series.astype()` and `DataFrame.astype()` now dispatch to `ExtensionArray.astype()`

(GH21185).

Slicing a single row of a DataFrame with multiple ExtensionArrays of the same type now preserves the dtype, rather than coercing to object (GH22784)
Bug when concatenating multiple Series with different extension dtypes not casting to object dtype (GH22994)

Series backed by an ExtensionArray now work with *util.hash__object()* (GH23066)

DataFrame.stack() no longer converts to object dtype for DataFrames where each column has the same extension dtype. The output Series will have the same dtype as the columns (GH23077).

Series.unstack() and *DataFrame.unstack()* no longer convert extension arrays to object-dtype ndarrays. Each column in the output DataFrame will now have the same dtype as the input (GH23077).

Bug when grouping *DataFrame.groupby()* and aggregating on ExtensionArray it was not returning the actual ExtensionArray dtype (GH23227).

Bug in *.merge()* when merging on an extension array-backed column (GH23020).

5.5. Version 0.24

Deprecations

- *MultiIndex.labels* has been deprecated and replaced by *MultiIndex.codes*. The functionality is unchanged. The new name better reflects the natures of these codes and makes the MultiIndex API more similar to the API for *CategoricalIndex* (GH13443). As a consequence, other uses of the name *labels* in MultiIndex have also been deprecated and replaced with *codes*:

You should initialize a MultiIndex instance using a parameter named *codes* rather than *labels*.

MultiIndex.set_labels has been deprecated in favor of *MultiIndex.set_codes()*.

For method `MultiIndex.copy()`, the `labels` parameter has been deprecated and replaced by a `codes` parameter.

`DataFrame.to_stata()`, `read_stata()`, `StataReader` and `StataWriter` have deprecated the `encoding` argument. The encoding of a Stata dta file is determined by the file type and cannot be changed (GH21244)

`MultiIndex.to_hierarchical()` is deprecated and will be removed in a future version (GH21613)

`Series.ptp()` is deprecated. Use `numpy.ptp` instead (GH21614)

`Series.compress()` is deprecated. Use `Series[condition]` instead (GH18262)

The signature of `Series.to_csv()` has been uniformed to that of `DataFrame.to_csv()`: the name of the first argument is now `path_or_buf`, the order of subsequent arguments has changed, the `header` argument now defaults to `True`. (GH19715)

`Categorical.from_codes()` has deprecated providing float values for the `codes` argument. (GH21767)

`.read_table()` is deprecated. Instead, use `read_csv()` passing `sep='\t'` if necessary. This deprecation has been removed in 0.25.0. (GH21948)

`Series.str.cat()` has deprecated using arbitrary list-likes *within* list-likes. A list-like container may still contain many `Series`, `Index` or 1-dimensional `np.ndarray`, or alternatively, only scalar values. (GH21950)

`FrozenNDArray.searchsorted()` has deprecated the `v` parameter in favor of `value` (GH14645)

`DatetimeIndex.shift()` and `PeriodIndex.shift()` now accept `periods` argument instead of `n` for consistency with `Index.shift()` and `Series.shift()`. Using `n` throws a deprecation warning (GH22458, GH22912)

The `fastpath` keyword of the different `Index` constructors is deprecated (GH23110).

`Timestamp.tz_localize()`, `DatetimeIndex.tz_localize()`, and `Series.tz_localize()` have deprecated the `errors` argument in favor of the nonexistent argument (GH8917)

The class `FrozenNDArray` has been deprecated. When unpickling, `FrozenNDArray` will be unpickled to `np.ndarray` once this class is removed (GH9031)

The methods `DataFrame.update()` and `Panel.update()` have deprecated the `raise_conflict=False|True` keyword in favor of `errors='ignore'|'raise'` (GH23585)

The methods `Series.str.partition()` and `Series.str.rpartition()` have deprecated the `pat` keyword in favor of `sep` (GH22676)

Deprecated the `nthreads` keyword of `.read_feather()` in favor of `use_threads` to reflect the changes in `pyarrow` $\geq 0.11.0$. (GH23053) `.read_excel()` has deprecated accepting `usecols` as an integer. Please pass in a list of ints from 0 to `usecols` inclusive instead (GH23527) Constructing a `TimedeltaIndex` from data with `datetime64`-typed data is deprecated, will raise `TypeError` in a future version (GH23539) Constructing a `DatetimeIndex` from data with `timedelta64`-typed data is deprecated, will raise `TypeError` in a future version (GH23675) The `keep_tz=False` option (the default) of the `keep_tz` keyword of `DatetimeIndex.to_series()` is deprecated (GH17832).

Timezone converting a tz-aware `datetime.datetime` or `Timestamp` with `Timestamp` and the `tz` argument is now deprecated. Instead, use `Timestamp.tz_convert()` (GH23579)

`.api.types.is_period()` is deprecated in favor of `.api.types.is_period_dtype` (GH23917)

`.api.types.is_datetimetz()` is deprecated in favor of `.api.types.is_datetime64tz` (GH23917)

Creating a `TimedeltaIndex`, `DatetimeIndex`, or `PeriodIndex` by passing `range` arguments `start`, `end`, and `periods` is deprecated in favor of `timedelta_range()`, `date_range()`, or `period_range()` (GH23919)

Passing a string alias like `'datetime64[ns, UTC]'` as the `unit` parameter to `DatetimeTZDtype` is deprecated. Use `DatetimeTZDtype.construct_from_string` instead (GH23990).

The `skipna` parameter of `infer_dtype()` will switch to `True` by default in a

future version of (GH17066, GH24050)

In *Series.where()* with Categorical data, providing an other that is not present in the categories is deprecated. Convert the categorical to a different dtype or add the other to the categories first (GH24077).

Series.clip_lower(), *Series.clip_upper()*, *DataFrame.clip_lower()* and *DataFrame.clip_upper()* are deprecated and will be removed in a future version. Use *Series.clip(lower=threshold)*, *Series.clip(upper=threshold)* and the equivalent *DataFrame* methods (GH24203)

Series.nonzero() is deprecated and will be removed in a future version (GH18262)

Passing an integer to *Series.fillna()* and *DataFrame.fillna()* with *timedelta64[ns]* dtypes is deprecated, will raise *TypeError* in a future version. Use *obj.fillna(pd.Timedelta(...))* instead (GH24694)

Series.cat.categorical, *Series.cat.name* and *Series.cat.index* have been deprecated. Use the attributes on *Series.cat* or *Series* directly. (GH24751).

Passing a dtype without a precision like *np.dtype('datetime64')* or *timedelta64* to *Index*, *DatetimeIndex* and *TimedeltaIndex* is now deprecated. Use the nanosecond-precision dtype instead (GH24753).

Integer addition/subtraction with datetimes and timedeltas is deprecated

In the past, users could—in some cases—add or subtract integers or integer-dtype arrays from *Timestamp*, *DatetimeIndex* and *TimedeltaIndex*.

This usage is now deprecated. Instead add or subtract integer multiples of the object's *freq* attribute (GH21939, GH23878).

Previous behavior :

5.5. Version 0.24

In [5]: `ts = pd.Timestamp('1994-05-06 12:15:16', freq=pd.offsets.Hour())` **In**


```

[6]: ts + 2
Out[6]: Timestamp('1994-05-06 14:15:16', freq='H')
In [7]: tdi = pd.timedelta_range('1D', periods=2)
In [8]: tdi - np.array([2, 1])
Out[8]: TimedeltaIndex(['-1 days', '1 days'], dtype='timedelta64[ns]',
freq=None) In [9]: dti = pd.date_range('2001-01-01', periods=2, freq='7D')
In [10]: dti + pd.Index([1, 2])
Out[10]: DatetimeIndex(['2001-01-08', '2001-01-22'],
dtype='datetime64[ns]', , _freq=None)

```

New behavior:

```

In [108]: ts = pd.Timestamp('1994-05-06 12:15:16', freq=pd.offsets.Hour())
In [109]: ts + 2 * ts.freq
Out[109]: Timestamp('1994-05-06 14:15:16', freq='H')
In [110]: tdi = pd.timedelta_range('1D', periods=2)
In [111]: tdi - np.array([2 * tdi.freq, 1 * tdi.freq])
Out[111]: TimedeltaIndex(['-1 days', '1 days'], dtype='timedelta64[ns]',
freq=None) In [112]: dti = pd.date_range('2001-01-01', periods=2, freq='7D')
In [113]: dti + pd.Index([1 * dti.freq, 2 * dti.freq])
Out[113]: DatetimeIndex(['2001-01-08', '2001-01-22'],
dtype='datetime64[ns]', , _freq=None)

```

Passing integer data and a timezone to DatetimeIndex

The behavior of *DatetimeIndex* when passed integer data and a timezone is changing in a future version of . Previously, these were interpreted as wall times in the desired timezone. In the future, these will be interpreted as wall times in UTC, which are then converted to the desired timezone (GH24559).

The default behavior remains the same, but issues a warning:

```

In [3]: pd.DatetimeIndex([946684800000000000], tz="US/Central")
/bin/ipython:1: FutureWarning:

```

Passing integer-dtype data and a timezone to *DatetimeIndex*. Integer values will be interpreted differently in a future version of . Previously, these were viewed as `datetime64[ns]` values representing the wall time *in the specified timezone*. In the future, these will be viewed as `datetime64[ns]` values

representing the wall time **in UTC**. This is similar to a nanosecond-precision UNIX epoch. To accept the future behavior, use

```
pd.to_datetime(integer_data, utc=True).tz_convert(tz)
```

To keep the previous behavior, use

```
pd.to_datetime(integer_data).tz_localize(tz)
```

```
#!/bin/python3
```

```
Out[3]: DatetimeIndex(['2000-01-01 00:00:00-06:00'], dtype='datetime64[ns, US/
```

```
, → Central]', freq=None) (continues on next page) (continued from previous page)
```

As the warning message explains, opt in to the future behavior by specifying that the integer values are UTC, and then converting to the final timezone:

```
In [114]: pd.to_datetime([9466848000000000000],  
utc=True).tz_convert('US/Central') Out[114]: Out[114]: 31 18:00:00-  
06:00'], dtype='datetime64[ns, US/
```

```
, → Central]', freq=None)
```

The old behavior can be retained with by localizing directly to the final timezone:

```
In [115]:
```

```
pd.to_datetime([9466848000000000000]).tz_localize('US/Central') Out[115]:  
DatetimeIndex(['2000-01-01 00:00:00-06:00'],  
dtype='datetime64[ns, US/
```

```
, → Central]', freq=None)
```

Converting timezone-aware Series and Index to NumPy arrays

The conversion from a *Series* or *Index* with timezone-aware datetime data will change to preserve timezones by default (GH23569). NumPy doesn't have a dedicated dtype for timezone-aware datetimes. In the past, converting a *Series* or

DatetimeIndex with timezone-aware datetimes would convert to a NumPy array by

converting the tz-aware data to UTC
dropping the timezone-info
returning a numpy.ndarray with datetime64[ns] dtype

Future versions of will preserve the timezone information by returning an object-dtype NumPy array where each value is a *Timestamp* with the correct timezone attached

In [116]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET")) **In**

[117]: ser **Out[117]:**

2000-01-01 00:00:00+01:00

2000-01-02 00:00:00+01:00

Length: 2, dtype: datetime64[ns, CET]

The default behavior remains the same, but issues a warning In [8]:

np.asarray(ser)

/bin/ipython:1: FutureWarning: Converting timezone-aware DatetimeArray to timezone

, → naive ndarray **with** 'datetime64[ns]' dtype. In the future, this will **return** an ndarray **with** 'object' dtype where each element **is** a '.Timestamp' **with** the correct

, → 'tz'.

To accept the future behavior, **pass** 'dtype=object'.

To keep the old behavior, **pass** 'dtype="datetime64[ns]"'.

#!/bin/python3 Out[8]:

array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'], dtype='datetime64[ns]')

The previous or future behavior can be obtained, without any warnings, by specifying the dtype

5.5. Version 0.24

Previous behavior

In [118]: np.asarray(ser, dtype='datetime64[ns]') **Out[118]:**

array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'], dtype='datetime64[ns]')

Future behavior

New behavior

In [119]: np.asarray(*ser*, dtype=object) **Out[119]:**

```
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),  
Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')], dtype=object)  
Or by using Series.to_numpy()
```

In [120]: *ser.to_numpy()* **Out[120]:**

```
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),  
Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')], dtype=object)
```

In [121]: *ser.to_numpy(dtype="datetime64[ns]")* **Out[121]:**

```
array(['1999-12-31T23:00:00.000000000', '2000-01-  
01T23:00:00.000000000'], dtype='datetime64[ns]')
```

All the above applies to a *DatetimeIndex* with tz-aware values as well.

Removal of prior version deprecations/changes

The LongPanel and WidePanel classes have been removed (GH10892)

Series.repeat() has renamed the *reps* argument to *repeats* (GH14645)

Several private functions were removed from the (non-public) module *.core.common* (GH22001)

Removal of the previously deprecated module *.core.datetools* (GH14105, GH14094)

Strings passed into *DataFrame.groupby()* that refer to both column and index levels will raise a *ValueError* (GH14432)

Index.repeat() and *MultiIndex.repeat()* have renamed the *n* argument to *repeats* (GH14645)

The *Series* constructor and *.astype* method will now raise a *ValueError* if timestamp dtypes are passed in without a unit (e.g. *np.datetime64*) for the dtype parameter (GH15987)

Removal of the previously deprecated *as_indexer* keyword completely from *str.match()* (GH22356, GH6581)

The modules *.types*, *.computation*, and *.util.decorators* have been removed (GH16157, GH16250)

Removed the `.formats.style` shim for `.io.formats.style.Styler` (GH16059)
`.pnow`, `.match`, `.groupby`, `pd.get_store`, `pd.Expr`, and `pd.Term` have been removed (GH15538, GH15940)

`Categorical.searchsorted()` and `Series.searchsorted()` have renamed the `v` argument to `value` (GH14645)
`.parser`, `.lib`, and `.tslib` have been removed (GH15537)
`Index.searchsorted()` have renamed the `key` argument to `value` (GH14645)
`DataFrame consolidate` and `Series consolidate` have been removed (GH15501)

Removal of the previously deprecated module `.json` (GH19944) The module `.tools` has been removed (GH15358, GH16005)

`SparseArray.get_values()` and `SparseArray.to_dense()` have dropped the `fill` parameter (GH14686)
`DataFrame.sortlevel` and `Series.sortlevel` have been removed (GH15099)
`SparseSeries.to_dense()` has dropped the `sparse_only` parameter (GH14686)
`DataFrame.astype()` and `Series.astype()` have renamed the `raise_on_error` argument to `errors` (GH14967)
`is_sequence`, `is_any_int_dtype`, and `is_floating_dtype` have been removed from `.api.types` (GH16163, GH16189)

Performance improvements

Slicing `Series` and `DataFrames` with an monotonically increasing `CategoricalIndex` is now very fast and has speed comparable to slicing with an `Int64Index`. The speed increase is both when indexing by label (using `.loc`) and position (`.iloc`) (GH20395) Slicing a monotonically increasing `CategoricalIndex` itself (i.e. `ci[1000:2000]`) shows similar speed improvements as above (GH21659)

Improved performance of `CategoricalIndex.equals()` when comparing to another `CategoricalIndex` (GH24023) Improved performance of `Series.describe()` in case of numeric dtypes (GH21274)
Improved performance of `.core.groupby.GroupBy.rank()` when dealing with tied rankings (GH21237)
Improved performance of `DataFrame.set_index()` with columns consisting of `Period` objects (GH21582, GH21606)

Improved performance of *Series.at()* and *Index.get_value()* for Extension Arrays values (e.g. *Categorical*) (GH24204)

Improved performance of membership checks in *Categorical* and *CategoricalIndex* (i.e. *x* in cat-style checks are much faster).
CategoricalIndex.contains() is likewise much faster (GH21369, GH21508)

Improved performance of *HDFStore.groups()* (and dependent functions like *HDFStore.keys()*). (i.e. *x* in store checks are much faster) (GH21372)

Improved the performance of *.get_dummies()* with *sparse=True* (GH21997)

Improved performance of *IndexEngine.get_indexer_non_unique()* for sorted, non-unique indexes (GH9466)

Improved performance of *PeriodIndex.unique()* (GH23083) Improved performance of *concat()* for Series objects (GH23404)