Intermediate Python

Obi Ike-Nwosu

Intermediate Python

Obi Ike-Nwosu

This book is for sale at http://leanpub.com/intermediatepython

This version was published on 2015-09-29

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

1. An Introduction

The Python Programming language has been around for quite a while. Development work was started on the first version of Python by Guido Van Rossum in 1989. Since then, it has grown to become a highly loved and revered language that has been used and continues to be used in a host of different application types.

The Python interpreter and the extensive standard library that come with the interpreter are available for free in source or binary form for all major platforms from the Python Web site. This site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter can easily be extended with new functions and data types implemented in C, C++ or any other language that is callable from C. Python is also suitable as an extension language for customisable applications. One of the most notable feature of python is the easy and white-space aware syntax.

This book is intended as a concise intermediate level treatise on the Python programming language. There is a need for this due to the lack of availability of materials for python programmers at this level. The material contained in this book is targeted at the programmer that has been through a beginner level introduction to the Python programming language or that has some experience in a different object oriented programming language such as Java and wants to gain a more in-depth understanding of the Python programming language in a holistic manner. It is not intended as an introductory tutorial for beginners although programmers with some experience in other languages may find the very short tutorial included instructive.

The book covers only a handful of topics but tries to provide a holistic and in-depth coverage of these topics. It starts with a short tutorial introduction to get the reader up to speed with the basics of Python; experienced programmers from other object oriented languages such as Java may find that this is all the introduction to Python that they need. This is followed by a discussion of the Python object model then it moves on to discussing object oriented programming in Python. With a firm understanding of the Python object model, it goes ahead to discuss functions and functional programming. This is followed by a discussion of meta-programming techniques and their applications. The remaining chapters cover generators, a complex but very interesting topic in Python, modules and packaging, and python runtime services. In between, intermezzos are used to discuss topics are worth knowing because of the added understanding they provide.

I hope the content of the book achieves the purpose for the writing of this book. I welcome all feedback readers may have and actively encourage readers to provide such feedback.

1.1 The Evolution of Python

In December 1989, Guido Van Rossum started work on a language that he christened **Python**. Guido Van Rossum had been part of the team that worked on the ABC programming language as part of the Amoeba operating systems in the 1980s at CWI (Centrum Wiskunde & Informatica) in Amsterdam and although he liked the ABC programming language, he was frustrated by a number of features or lack of thereof. Guido wanted a high level programming language that would speed up the development of utilities for the the Amoeba project and ABC was not the answer. The ABC programming language would however play a very influential role in the development of python as Guido took parts he liked from the language and provided solutions for aspects of the ABC programming language that he found frustrating.

Guido released the first version of the Python programming language in February 1991. This release was object oriented, had a module system, included exception handling, functions, and the core data types of `list`, `dict`, `str` and others. Python version 1.0 was released in January 1994 and this release included functional programming constructs such as `lambda`, `map`, `filter` and `reduce`.

Python 1.2 was the last version released while Guido was still at CWI. In 1995, Van Rossum continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the language with indirect funding support from DARPRA.

By version 1.4, Python had acquired several new features including the *Modula-3* inspired keyword arguments and built-in support for complex numbers. It also included a basic form of data hiding by name mangling. Python 1.5 was released on December 31, 1997 while python 1.6 followed on September 5, 2000.

Python 2.0 was released on October 16, 2000 and it introduced list comprehensions, a feature borrowed from the functional programming languages *SETL* and *Haskell* as well as a garbage collection system capable of collecting reference cycles.

Python 2.2 was the first major update to the Python type system. This update saw the unification of Python's in-built types and user defined classes written in Python into one hierarchy. This single unification made Python's object model purely and consistently object oriented. This update to the class system of Python added a number of features that improved the programming experience. These included:

1. The ability to subclass in built types such as `dicts` and `lists`.
2. The addition of static and class methods
3. The addition of properties defined by `get` and `set` methods.
4. An update to meta-classes, `__new__()` and `super()` methods, and MRO algorithms.

The next major milestone was Python 3 released on December 2, 2008. This was designed to rectify certain fundamental design flaws in the language that could not be implemented while maintaining full backwards compatibility with the 2.x series.

1.2 Python 2 *vs* Python 3

Perhaps the most visible and disruptive change to the Python ecosystem has been the introduction of Python 3. The major changes introduced into Python 3 include the

following:

1. `print()` is now a function
2. Some well known python *APIs* such as `range()`, `dict.keys()`, `dict.values()` return views and iterators rather than lists improving efficiency when they are used.
3. The rules for ordering comparisons have been simplified. For example, a heterogeneous list cannot be sorted, because all the elements of a list must be comparable to each other.
4. The integer types have been whittled down to only one, i.e. `int`. `long` is also an `int`.
5. The division of two integers returns a float instead of an integer. `//` can be used to return an integer when division takes place.
6. All texts are now unicode but encoded unicode text is represented as binary data and any attempt to mix text and data will result in an exception. This breaks backwards compatibility with python 2.x versions.
7. Python 3 also saw the introduction of some new syntax such as function annotations, the `nonlocal` statement, extended iterable unpacking, set literals, dictionary comprehensions etc.
8. Python 3 also saw update to some syntax such as that of exception handling, meta-class specification, list comprehensions etc.

The full details on changes from python 2 to python 3 can be viewed on the python website. The rest of the book will assumes the use of Python 3.4.

1.3 The Python Programming Language

The *Python programming language* refers to the language as documented in the language reference. There is no official language specification but the language reference provides enough details to guide anyone implementing the Python programming language. The implementation of the Python programming language available on the Python website is an implementation written in `C` and commonly referred to as *CPython*. This is normally used as the reference implementation. However, there are other Python implementations in different languages. Popular among these are *PyPy: python implemented in python* and *Jython: python implemented in Java*. For the rest of this book, the reference *CPython* version that is freely distributed through the Python website is used.

2. A Very Short Tutorial

This short tutorial introduces the reader informally to the basic concepts and features of the Python programming language. It is not intended to be a comprehensive introductory tutorial to programming for a complete novice but rather assumes the reader has had previous experience programming with an object oriented programming language.

2.1 Using Python

Python is installed by default on most Unix-based systems including the Mac OS and various Linux-based distributions. To check if python is installed, open the command-line and type `python`. If not installed then python can be installed by visiting the python language website and following instructions on how to install python for the given platform.

The Python Interpreter

The python interpreter can be invoked by opening the command-line and typing in `python`. In the case that it has been installed but the command cannot be found then the full path to the installation should be used or added to the path. Invoking the interpreter using `python` brings up the python interactive session with an `REPL` prompt. The primary prompt, `>>>`, signals a user to enter statements while the secondary prompt, `...`, signals a continuation line as shown in the following example.

```
>>> def hello():
...       print("Hello world")
...
>>>
```

A user can type in python statements at the interpreter prompt and get instant feedback. For example, we can evaluate expressions at the `REPL` and get values for such expressions as in the following example.

```
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> var_1 = 3
>>> var_2 = 3
>>> var_1*var_2
9
>>>
```

Typing `Ctrl-D` at the primary prompt causes the interpreter to exit the session.

2.2 Python Statements, Line Structure and Indentation

A program in Python is composed of a number of logical lines and each of these logical lines is delimited by the `NEWLINE` token. Each logical line is equivalent to a valid

statement. Compound statements however can be made up of multiple logical lines. A logical line is made from one or more physical lines using the explicit or implicit line joining rules. A physical line is a sequence of characters terminated by an `end-of-line` sequence. Python implicitly sees physical lines as logical lines eliminating the explicit need for semi-colons in terminating statements as in Java. Semi-colons however play a role in python; it is possible to have multiple logical lines on the same physical line by separating the logical lines with semi-colons such as shown below:

```
>>> i = 5; print i;
5
```

Multiple physical lines can be explicitly joined into a single logical line by use of the line continuation character, `\`, as shown below:

```
>>> name = "Obi Ike-Nwosu"
>>> cleaned_name = name.replace("-", " "). \
...                     replace(" ", "")
>>> cleaned_name
'ObiIkeNwosu'
>>>
```

Lines are joined implicitly, thus eliminating the need for line continuation characters, when expressions in triple quoted strings, enclosed in parenthesis (…), brackets [….] or braces {…} spans multiple lines.

From discussions above, it can be inferred that there are two types of statements in python:

1. Simple statements that span a single logical line. These include statements such as assignment statements, yield statements etc. A simple statement can be summarized as follows:

```
simple_stmt ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | global_stmt
                | nonlocal_stmt

::= means is defined as
| means or
```

1. Compound statements that span multiple logical lines statements. These include statements such as the `while` and `for` statements. A compound statement is summarized as thus in python:

```
compound_stmt ::=  if_stmt
                   | while_stmt
                   | for_stmt
                   | try_stmt
                   | with_stmt
                   | funcdef
                   | classdef
suite           ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=  stmt_list NEWLINE | compound_stmt
stmt_list       ::=  simple_stmt (";" simple_stmt)* [";"]
```

Compound statements are made up of one or more *clauses*. A *clause* is made up of a *header* and a *suite*. The clause headers for a given compound statement are all at the same indentation level; they begin with a unique identifier, *while*, *if* etc., and end with a colon. The suite execution is controlled by the header. This is illustrate with the example below:

```
>>> num = 6
# if statement is a compound statement
     # clause header controls execution of indented block that follows
>>> if num % 2 == 0:
          # indented suite block
...       print("The number {} is even".format(num))
...
The number 6 is even
>>>
```

The suite may be a set of one or more statements that follow the header's colon with each statement separated from the previous by a semi-colon as shown in the following example.

```python
>>> x = 1
>>> y = 2
>>> z = 3
>>> if x < y < z: print(x); print(y); print(z)
...
1
2
3
```

The suite is conventionally written as one or more **indented** statements on subsequent lines that follow the header such as below:

```python
>>> x = 1
>>> y = 2
>>> z = 3
>>> if x < y < z:
...     print(x)
...     print(y);
...     print(z)
...
1
2
3
```

Indentations are used to denote code blocks such as function bodies, conditionals, loops and classes. Leading white-space at the beginning of a logical line is used to compute the indentation level for that line, which in turn is used to determine the grouping of statements. Indentation used within the code body must always match the indentation of the the first statement of the block of code.

2.3 Strings

Strings are represented in Python using double `"..."` or single `'...'` quotes. Special characters can be used within a string by escaping them with `\` as shown in the following example:

```
# the quote is used as an apostrophe so we escape it for python to
# treat is as an apostrophe rather than the closing quote for a string
>>> name = 'men\'s'
>>> name
"men's"
>>>
```

To avoid the interpretation of characters as special characters, the character, `r`, is added before the opening quote for the string as shown in the following example.

```
>>> print('C:\some\name')   # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')   # note the r before the quote
C:\some\name
```

String literals that span multiple lines can be created with the triple quotes but newlines are automatically added at the end of a line as shown in the following snippet.

```
>>> para = """hello world I am putting together a
... book for beginners to get to the next level in python"""
# notice the new line character
>>> para
'hello world I am putting together a \nbook for beginners to get to the next level in python'
# printing this will cause the string to go on multiple lines
>>> print(para)
hello world I am putting together a
book for beginners to get to the next level in python
>>>
```

To avoid the inclusion of a newline, the continuation character `\` should be used at the end of a line as shown in the following example.

```
>>> para = """hello world I am putting together a \
... book for beginners to get to the next level in python"""
>>> para
'hello world I am putting together a book for beginners to get to the next level in python'
>>> print(para)
hello world I am putting together a book for beginners to get to the next level in python
>>>
```

String are immutable so once created they cannot be modified. There is no `character` type so characters are assumed to be strings of length, `1`. Strings are sequence types so support sequence type operations except assignment due to their immutability. Strings can be indexed with integers as shown in the following snippet:

```
>>> name = 'obiesie'
>>> name[1]
'b'
>>>
```

Strings can be concatenated to create new strings as shown in the following example

```
>>> name = 'obiesie'
>>> surname = " Ike-Nwosu"
>>> full_name = name + surname
>>> full_name
'obiesie Ike-Nwosu'
>>>
```

One or more string literals can be concatenated together by writing them next to each other as shown in the following snippet:

```
>>> 'Py' 'thon'
'Python'
>>>
```

The built-in method `len` can also be used to get the length of a string as shown in the following snippet.

```
>>> name = "obi"
>>> len(name)
3
>>>
```

2.4 Flow Control

`if-else` and `if-elif-else` statements

Python supports the `if` statement for conditional execution of a code block.

```
>>> name = "obi"
>>> if name == "obi":
...      print("Hello Obi")
...
Hello Obi
>>>
```

The `if` statement can be followed by zero or more `elif` statements and an optional `else` statement that is executed when none of the conditions in the `if` or `elif` statements have been met.

```
>>> if name == "obi":
...      print("Hello Obi")
... elif name == "chuks":
...      print("Hello chuks")
```

```
... else:
...     print("Hello Stranger")
Hello Stranger
>>>
```

for and range statements

The `while` and `for` statements constitute the main looping constructs provided by python.

The `for` statement in python is used to iterate over sequence types (lists, sets, tuples etc.). More generally, the `for` loop is used to iterate over any object that implements the python *iterator* protocol. This will be discussed further in chapters that follow. Example usage of the `for` loop is shown by the following snippet:

```
>>> names = ["Joe", "Obi", "Chris", "Nkem"]
>>> for name in names:
...     print(name)
...
Joe
Obi
Chris
Nkem
>>>
```

Most programming languages have a syntax similar to the following for iterating over a progression of numbers:

```
for(int x = 10; x < 20; x = x+1) {
          // do something here
      }
```

Python replaces the above with the simpler `range()` statement that is used to generate an arithmetic progression of integers. For example:

```
>>> for i in range(10, 20):
...     print i
...
10
11
12
13
14
15
16
17
18
19
>>>
```

The `range` statement has a syntax of `range(start, stop, step)`. The stop value is never part of the progression that is returned.

while statement

The `while` statement executes the statements in its suite as long as the condition expression in the `while` statement evaluates to `true`.

```
>>> counter = 10
>>> while counter > 0: # the conditional expression is 'counter>0'
...       print(counter)
...       counter = counter - 1
...
10
9
8
7
6
5
4
3
2
1
```

break and continue statements

The `break` keyword is used to escape from an enclosing loop. Whenever the `break` keyword is encountered during the execution of a loop, the loop is abruptly exited and no other statement within the loop is executed.

```
>>> for i in range(10):
...       if i == 5:
...                 break
...       else:
...                 print(i)
...
0
1
2
3
4
```

The `continue` keyword is used to force the start of the next iteration of a loop. When used the interpreter ignores all statements that come after the `continue` statement and continues with the next iteration of the loop.

```
>>> for i in range(10):
          # if i is 5 then continue so print statement is ignored and the next iteration
          # continues with i set to 6
...       if i == 5:
...                 continue
...       print("The value is " + str(i))
...
The value is 0
The value is 1
The value is 2
The value is 3
The value is 4
# no printed value for i == 6
The value is 6
The value is 7
The value is 8
The value is 9
```

In the example above, it can be observed that the number 5 is not printed due to the use of `continue` when the value is 5 however all subsequent values are printed.

`else` clause with looping constructs

Python has a quirky feature in which the `else` keyword can be used with looping constructs. When an `else` keyword is used with a looping construct such as `while` or `for`, the statements within the *suite* of the `else` statement are executed as long as the looping construct was not ended by a `break` statement.

```
# loop exits normally
>>> for i in range(10):
...     print(i)
... else:
...     print("I am in quirky else loop")
...
0
1
2
3
4
5
6
7
8
9
I am in quirky else loop
>>>
```

If the loop was exited by a `break` statement, the execution of the suite of the `else` statement is skipped as shown in the following example:

```
>>> for i in range(10):
...     if i == 5:
...         break
...     print(i)
... else:
...     print("I am in quirky else loop")
...
0
1
2
3
4
>>>
```

Enumerate

Sometimes, when iterating over a list, a tuple or a sequence in general, having access to the index of the item, as well as the item being enumerated over maybe necessary. This could achieved using a `while` loop as shown in the following snippet:

```
>>> names = ["Joe", "Obi", "Chris", "Jamie"]
>>> name_count = len(names)
>>> index = 0
>>> while index < name_count:
...     print("{}. {}".format(index, names[index]))
...     index = index + 1
```

```
...
0. Joe
1. Obi
2. Chris
3. Jamie
```

The above solution is how one would go about it in most languages but python has a better alternative to such in the form of the `enumerate` keyword. The above solution can be reworked beautifully in python as shown in the following snippet:

```
>>> for index, name in enumerate(names):
...     print("{}. {}".format(index, name))
...
0. Joe
1. Obi
2. Chris
3. Jamie
>>>
```

2.5 Functions

Named functions are defined with the `def` keyword which must be followed by the function name and the parenthesized list of formal parameters. The `return`keyword is used to return a value from a function definition. A python function definition is shown in the example below:

```
def full_name(first_name, last_name):
    return " ".join((first_name, last_name))
```

Functions are invoked by calling the function name with required arguments in parenthesis for example `full_name("Obi", "Ike-Nwosu")`. Python functions can return multiple values by returning a tuple of the required values as shown in the example below in which we return the quotient and remainder from a division operation:

```
>>> def divide(a, b):
...     return divmod(q, r)
...
>>> divide(7, 2)
(3, 1)
>>>
```

Python functions can be defined without `return` keyword. In that case the default returned value is `None` as shown in the following snippet:

```
>>> def print_name(first_name, last_name):
...     print(" ".join((first_name, last_name)))
...
>>> print_name("Obi", "Ike-Nwosu")
Obi Ike-Nwosu
>>> x = print_name("Obi", "Ike-Nwosu")
Obi Ike-Nwosu
>>> x
>>> type(x)
<type 'NoneType'>
>>>
```

The `return` keyword does not even have to return a value in python as shown in the following example.

```
>>> def dont_return_value():
...       print("How to use return keyword without a value")
...       return
...
>>> dont_return_value()
How to use return keyword without a value
```

Python also supports anonymous functions defined with the `lambda` keyword. Python's lambda support is rather limited, crippled a few people may say, because it supports only a single expression in the body of the lambda expression. Lambda expressions are another form of syntactic sugar and are equivalent to conventional named function definition. An example of a lambda expression is the following:

```
>>> square_of_number = lambda x: x**2
>>> square_of_number
<function <lambda> at 0x101a07158>
>>> square_of_number(2)
4
>>>
```

2.6 Data Structures

Python has a number of built-in data structures that make programming easy. The built-in data structures include lists, tuples and dictionaries.

1. Lists: Lists are created using square brackets, `[]` or the `list()` function. The empty list is denoted by `[]`. Lists preserve the order of items as they are created or insert into the list. Lists are sequence types so support integer indexing and all other sequence type subscripting that will be discussed in chapters that follow. Lists are indexed by integers starting with zero and going up to the length of the list minus one.

   ```
   >>> name = ["obi", "ike", "nwosu"]
   >>> name[0]
   'obi'
   >>>
   ```

Items can be added to a list by appending to the list.

```
>>>       name = ["obi", "ike", "nwosu"]
>>> name.append("nkem")
>>> names
["obi", "ike", "nwosu", "nkem"]

Elements can also be added to other parts of a list not just the end using `insert` method.

>>> name = ["obi", "ike", "nwosu"]
>>> name.insert(1, "nkem")
>>> names
["obi", "nkem", "ike", "nwosu"]
```

Two or more lists can be concatenated together with the `+` operator.

```
>>> name = ["obi", "ike", "nwosu"]
>>> name1 = ["James"]
>>> name + name1
["obi", "ike", "nwosu", "James"]
```

To get a full listing of all methods of the list, run the `help` command with `list` as argument.

1. Tuples: These are also another type of sequence structures. A tuple consists of a number of comma separated objects for example.

```
>>> companies = "Google", "Microsoft", "Tesla"
>>> companies
('Google', 'Microsoft', 'Tesla')
>>>
```

When defining a non-empty tuple the parenthesis is optional but when the tuple is part of a larger expression, the parenthesis is required. The parenthesis come in handy when defining an empty tuple for instance:

```
>>> companies = ()
>>> type(companies)
<class 'tuple'>
>>>
```

Tuples have a quirky syntax that some people may find surprising. When defining a single element tuple, the comma must be included after the single element regardless of whether or not parenthesis are included. If the comma is left out then the result of the expression is not a tuple. For instance:

```
>>> company = "Google",
>>> type(company)
<class 'tuple'>
>>>
>>> company = ("Google",)
>>> type(company)
<class 'tuple'>
# absence of the comma returns the value contained within the parenthesis
>>> company = ("Google")
>>> company
'Google'
>>> type(company)
<class 'str'>
>>>
```

Tuples are integer indexed just like lists but are immutable; once created the contents cannot be changed by any means such as by assignment. For instance:

```
>>> companies = ("Google", "Microsoft", "Palantir")
>>> companies[0]
'Google'
>>> companies[0] = "Boeing"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

However, if the object in a tuple is a mutable object such as a list, such object can be changed as shown in the following example:

```
>>> companies = (["lockheedMartin", "Boeing"], ["Google", "Microsoft"])
>>> companies
(['lockheedMartin', 'Boeing'], ['Google', 'Microsoft'])
>>> companies[0].append("SpaceX")
>>> companies
(['lockheedMartin', 'Boeing', 'SpaceX'], ['Google', 'Microsoft'])
>>>
```

1. Sets: A set is an unordered collection of objects that does not contain any duplicates. An empty set is created using `set()` or by using curly braces, `{}`. Sets are unordered so unlike tuples or lists they cannot be indexed by integers. However sets, with the exception of frozen sets, are mutable so one can `add`, `update` or `remove` from a set as shown in the following:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> basket_set = set()
>>> basket_set
set()
>>> basket_set.update(basket)
>>> basket_set
{'pear', 'orange', 'apple', 'banana'}
>>> basket_set.add("clementine")
>>> basket_set
{'pear', 'orange', 'apple', 'banana', 'clementine'}
>>> basket_set.remove("apple")
>>> basket_set
{'pear', 'orange', 'banana', 'clementine'}
>>>
```

2. Dictionary: This is a mapping data structure that is commonly referred to as an *associative array* or a *hash table* in other languages. Dictionaries or `dicts` as they are commonly called are indexed by keys that must be immutable. A pair of braces, `{...}` or method `dict()` is used to create a `dict`. Dictionaries are unordered set of `key:value` pairs, in which the keys are unique. A dictionary can be initialized by placing a set of `key:value` pairs within the braces as shown in the following example.

```
ages = {"obi": 24,
          "nkem": 23,
          "Chris": 23
          }
```

   The primary operations of interest that are offered by dictionaries are the storage of a value by the key and retrieval of stored values also by key. Values are retrieved by using indexing the dictionary with the key using square brackets as shown in the following example.

```
>>> ages["obi"]
24
```

Dictionaries are mutable so the values indexed by a key can be changed, keys can be deleted and added to the `dict`.

Python's data structures are not limited to just those listed in this section. For example the `collections` module provides additional data structures such as `queues` and `deques` however the data structures listed in this section form the workhorse for most Python applications. To get better insight into the capabilities of a data structure, the `help()` function is used with the name of the data structure as argument for example, `help(list)`.

2.7 Classes

The `class` statement is used to define new types in python as shown in the following example:

```
class Account:
    # class variable that is common to all instances of a class
    num_accounts = 0

    def __init__(self, name, balance):
        # start of instance variable
            self.name = name
            self.balance = balance
            # end of instance variables
            Account.num_accounts += 1

    def deposit(self, amt):
            self.balance = self.balance + amt

    def withdraw(self, amt):
            self.balance = self.balance - amt

    def inquiry(self):
            return "Name={}, balance={}".format(self.name, self.balance)

    @classmethod
    def from_dict(cls, params):
            params_dict = json.loads(params)
            return cls(params_dict.get("name"), params_dict.get("balance"))
```

Classes in python just like classes in other languages have class variables, instance variables, class methods, static methods and instance methods. When defining classes, the base classes are included in the parenthesis that follows the class name. For those that are familiar with Java, the `__init__` method is something similar to a constructor; it is in this method that instance variables are initialized. The above defined class can be initialized by *calling* the defined class with required arguments to `__init__` in parenthesis ignoring the `self` argument as shown in the following example.

```
>>> acct = Account("obie", 10000000)
```

Methods in a class that are defined with `self` as first argument are instance methods. The `self` argument is similar to `this` in java and refers to the object instance. Methods are called in python using the dot notation syntax as shown below:

```
>>> acct = Account("obie", 10000000)
>>>account.inquiry()
Name=obie, balance=10000000
```

Python comes with built-in function, `dir`, for introspection of objects. The `dir` function can be called with an object as argument and it returns a list of all attributes, methods and variables, of a class.

2.8 Modules

Functions and classes provide mean for structuring your Python code but as the code grows in size and complexity, there is a need for such code to be split into multiple files with each source file containing related definitions. The source files can then be *imported* as needed in order to access definitions in any of such source file. In python, we refer to source files as modules and modules have the `.py` extensions.

For example, the `Account` class definition from the previous section can be saved to a module called `Account.py`. To use this module else where, the `import` statement is used to import the module as shown in the following example:

```
>>> import Account
>>> acct = Account.Account("obie", 10000000)
```

Note that the import statement takes the name of the module without the `.py` extension. Using the import statement creates a name-space, in this case the `Account` name-space and all definitions in the module are available in such name-space. The dot notation (`.`) is used to access the definitions as required. An alias for an imported module can also be created using the `as` keyword so the example from above can be reformulated as shown in the following snippet:

```
>>> import Account as acct
>>> account = acct.Account("obie". 10000000)
```

It is also possible to import only the definitions that are needed from the module resulting in the following:

```
>>> from Account import Account
>>> account = Account("obie", 10000000)
```

All the definitions in a module can also be imported by using the wild card symbol a shown below:

```
>>> from Account import *
```

This method of imports is not always advised as it can result in name clashes when one of the name definitions being imported is already used in the current name-space. This is avoided by importing the module as a whole. Modules are also objects in Python so we

can introspect on them using the `dir` introspection function. Python modules can be further grouped together into packages. Modules and packages are discussed in depth in a subsequent chapter that follows.

2.9 Exceptions

Python has support for exceptions and exception handling. For example, when an attempt is made to divide by zero, a `ZeroDivisionError` is thrown by the python interpreter as shown in the following example.

```
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

During the execution of a program, an exception is raised when an error occurs; if the exception is not handled, a trace-back is dumped to the screen. Errors that are not handled will normally cause an executing program to terminate.

Exceptions can be handled in Python by using the `try…catch` statements. For example, the divide by zero exception from above could be handled as shown in the following snippet.

```
>>> try:
...     2/0
... except ZeroDivisionError as e:
...     print("Attempting to divide by 0. Not allowed")
...
Attempting to divide by 0. Not allowed
>>>
```

Exceptions in python can be of different types. For example, if an attempt was made to catch an `IOError` in the previous snippet, the program would terminate because the resulting exception type is a `ZeroDivisionError` exception. To catch all types of exceptions with a single handler, `try…catch Exception` is used but this is advised against as it becomes impossible to tell what kind of exception has occurred thus masking the exception

Custom exceptions can be defined to handle custom exceptions in our code. To do this, define a custom exception class that inherits from the `Exception` base class.

2.10 Input and Output

Python as expected has support for reading and writing to and from input and output sources. The file on the hard drive is the most popular IO device. The content of a file can be opened and read from using the snippet below:

```
f = open("afile.txt")
line = f.readline()
while line:
    print(line)
    line = f.readline()
```

The open method returns a file object or throws an exception if the file does not exist. The file object supports a number of methods such as `read` that reads the whole content of the file into a string or `readline` that reads the contents of the file one line at a time. Python supports the following syntactic sugar for iterating through the lines of a file.

```
for line in open("afile.txt"):
     print(line)
```

Python supports writing to a file as shown below:

```
f = open("out.txt", "w")
contents = ["I", "love", "python"]
for content in contents:
     f.write(content)
f.close()
```

Python also has support for writing to standard input and standard output. This can be done using the `sys.stdout.write()` or the `sys.stdin.readline()` from the `sys` module.

2.11 Getting Help

The python programming language has a very detailed set of documentation that can be obtained at the interpreter prompt by using the `help` method. To get more information about a syntactic construct or data structure, pass it as an argument to the help function for example `help(list)`.

3. Intermezzo: Glossary

A number of terms and esoteric python functions are used throughout this book and a good understanding of these terms is integral to gaining a better. and deeper understanding of python. A description of these terms and functions is provided in the sections that follow.

3.1 Names and Binding

In python, objects are referenced by *names*. `names` are analogous to variables in `C++` and `Java`.

```
>>> x = 5
```

In the above, example, `x` is a name that references the object, `5`. The process of *assigning* a reference to `5` to `x` is called *binding*. A binding causes a name to be associated with an object in the innermost scope of the currently executing program. Bindings may occur during a number of instances such as during variable assignment or function or method call when the supplied parameter is bound to the argument. It is important to note that names are just symbols and they have no *type* associated with them; **names are just references to objects that actually have types**

3.2 Code Blocks

A code block is a piece of program code that is executed as a single unit in python. Modules, functions and classes are all examples of code blocks. Commands typed in interactively at the REPL, script commands run with the `-c` option are also code blocks. A code block has a number of name-spaces associated with it. For example, a module code block has access to the `global`name-space while a function code block has access to the `local` as well as the `global` name-spaces.

3.3 Name-spaces

A *name-space* as the name implies is a context in which a given set of names is bound to objects. name-spaces in python are currently implemented as dictionary mappings. The *built-in* name-space is an example of a name-space that contains all the built-in functions and this can be accessed by entering `__builtins__.__dict__` at the terminal (the result is of a considerable amount). The interpreter has access to multiple name-spaces including *the global name-space*, *the built-in name-space* and *the local name-space*. name-spaces are created at different times and have different lifetimes. For example, a new local name-space is created at the start of a function execution and this name-space is discarded when the function exits or returns. The *global name-space* refers to the module wide name-space and all names defined in this name-space are available module-wide. The *local name-space* is created by function definitions while the *built-in name-space* contains all

the built-in names. These three name-spaces are the main name-space available to the interpreter.

3.4 Scopes

A scope is an area of a program in which a set of name bindings (name-spaces) is visible and directly accessible. Direct access is an important characteristic of a scope as will be explained when classes are discussed. This simply means that a name, `name`, can be used as is, without the need for dot notation such as `SomeClassOrModule.name` to access it. At runtime, the following scopes may be available.

1. Inner most scope with local names
2. The scope of enclosing functions if any (this is applicable for nested functions)
3. The current module's globals scope
4. The scope containing the builtin name-space.

Whan a name is used in python, the interpreter searches the name-spaces of the scopes in ascending order as listed above and if the name is not found in any of the name-spaces, an exception is raised. Python supports static scoping also known as lexical scoping; this means that the visibility of a set of name bindings can be inferred by only inspecting the program text.

Note

Python has a quirky scoping rule that prevents a reference to an object in the *global* scope from being modified in a local scope; such an attempt will throw an `UnboundLocalError` exception. In order to modify an object from the global scope within a local scope, the `global` keyword has to be used with the object name before modification is attempted. The following example illustrates this.

```
>>> a = 1
>>> def inc_a(): a += 2
...
>>> inc_a()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in inc_a
UnboundLocalError: local variable 'a' referenced before assignment
```

In order to modify the object from the global scope, the `global` statement is used as shown in the following snippet.

```
>>> a = 1
>>> def inc_a():
...       global a
...       a += 1
...
>>> inc_a()
>>> a
2
```

Python also has the `nonlocal` keyword that is used when there is a need to modify a variable bound in an outer non-global scope from an inner scope. This proves very handy when working with nested functions (also referred to as closures). A very trivial illustration of the `nonlocal` keyword in action is shown in the following snippet that defines a simple counter object that counts in ascending order.

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count # nonlocal captures the count binding from enclosing scope not glol
scope
...         count += 1
...         return count
...     return counter
...
>>> counter_1 = make_counter()
>>> counter_2 = make_counter()
>>> counter_1()
1
>>> counter_1()
2
>>> counter_2()
1
>>> counter_2()
2
```

3.5 `eval()`

`eval` is a python built-in method for dynamically executing python expressions in a string (the content of the string must be a valid python expression) or code objects. The function has the following signature `eval(expression, globals=None, locals=None)`. If supplied, the `globals` argument to the `eval` function must be a dictionary while the `locals` argument can be any mapping. The evaluation of the supplied expression is done using the *globals* and *locals* dictionaries as the global and local name-spaces. If the `__builtins__` is absent from the globals dictionary, the current globals are copied into globals before expression is parsed. This means that the expression will have either full or restricted access to the standard built-ins depending on the execution environment; this way the exection environment of `eval` can be restricted or *sandboxed*. `eval` when called returns the result of executing the expression or code object for example:

```python
>>> eval("2 + 1") # note the expression is in a string
3
```

Since `eval` can take arbitrary code obects as argument and return the value of executing such expressions, it along with `exec`, is used in executing arbitrary Python code that has been compiled into code objects using the `compile` method. Online Python interpreters are able to execute python code supplied by their users using both `eval` and `exec` among other methods.

3.6 `exec()`

`exec` is the counterpart to `eval`. This executes a string interpreted as a suite of python statements or a code object. The code supplied is supposed to be valid as file input in both cases. `exec` has the following signature: `exec(object[, globals[, locals]])`. The following is an example of `exec` using a string and the current name-spaces.

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
# the acct.py file is located somewhere on file
>>> cont = open('acct.py', 'r').read()
>>> cont
'class Account:\n    """base class for representing user accounts"""\n    num_accounts = 0\n\n    de\
f __init__(self, name, balance):\n        self.name = name \n        self.balance = balance \n      \
  Account.num_accounts += 1\n\n    def del_account(self):\n        Account.num_accounts -= 1\n\n    \
def __getattr__(self, name):\n        """handle attribute reference for non-existent attribute"""\n \
       return "Hey I dont see any attribute called {}".format(name)\n\n    def deposit(self, amt):\n\
        self.balance = self.balance + amt \n\n    def withdraw(self, amt):\n        self.balance = s\
elf.balance - amt \n\n    def inquiry(self):\n        return "Name={}, balance={}".format(self.name,\
 self.balance) \n\n'
>>> exec(cont)
# exec content of file using the default name-spaces
>>> Account # we can now reference the account class
<class '__main__.Account'>
>>>
```

In all instances, if optional arguments are omitted, the code is executed in the current scope. If only the `globals` argument is provided, it has to be a dictionary, that is used for both the global and the local variables. If `globals` and `locals` are given, they are used for the global and local variables, respectively. If provided, the `locals` argument can be any mapping object. If the `globals` dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. One can control the `builtins` that are available to the executed code by inserting custom `__builtins__` dictionary into `globals` before passing it to `exec()` thus creating a sandbox.

4. Objects 201

Python **objects** are the basic abstraction over data in python; *every value is an object* in python. Every object has an identity, a type and a value. An object's identity never changes once it has been created. The `id(obj)` function returns an integer representing the `obj's` identity. The `is` operator compares the identity of two objects returning a boolean. In *CPython*, the `id()` function returns an integer that is a memory location for the object thus uniquely identifying such object. This is an implementation detail and implementations of Python are free to return whatever value uniquely identifies objects within the interpreter.

The `type()` function returns an object's type; the type of an object is also an object itself. An object's type is also *normally* unchangeable. An object's type determines the operations that the object supports and also defines the possible values for objects of that type. Python is a dynamic language because types are not associated with variables so a variable, `x`, may refer to a string and later refer to an integer as shown in the following example.

```
x = 1
x = "Nkem"
```

However, Python unlike dynamic languages such as Javascript is strongly typed because the interpreter will never change the type of an object. This means that actions such as adding a string to a number will cause an exception in Python as shown in the following snippet:

```
>>> x = "Nkem"
>>> x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

This is unlike Javascript where the above succeeds because the interpreter implicitly converts the integer to a string then adds it to the supplied string.

Python objects are either one of the following:

1. Mutable objects: These refer to objects whose value can change. For example a list is a mutable data structure as we can grow or shrink the list at will.

    ```
    >>> x = [1, 2, 4]
    >>> y = [5, 6, 7]
    >>> x = x + y
    >>> x
    [1, 2, 4, 5, 6, 7]
    >>>
    ```

Programmers new to Python from other languages may find some behavior of mutable object puzzling; Python is a *pass-by-object-reference* language which means that the values of object references are the values passed to function or method calls and names bound to variables refer to these reference values. For example consider the snippets shown in the following example.

```
>>> x
[1, 2, 3]
# now x and y refer to the same list
>>> y = x
# a change to x will also be reflected in y
>>> x.extend([4, 5, 6])
>>> y
[1, 2, 3, 4, 5, 6]
```

`y` and `x` refer to the same object so a change to `x` is reflected in `y`. To fully understand why this is so, it must be noted that the variable, `x` does not actually hold the list, `[1, 2, 3]`, rather it holds a reference that points to the location of that object so when the variable, `y` is bound to the value contained in `x`, it now also contains the reference to the original list, `[1, 2, 3]`. Any operation on `x` finds the list that `x` refers to and carries out the operation on the list; `y` also refers to the same list thus the change is also reflected in the variable, `y`.

1. Immutable objects: These objects have values that cannot be changed. A `tuple` is an example of an immutable data structure because once created we can not change the constituent objects as shown below:

    ```
    >>> x = (1, 2, 3, 4)
    >>> x[0]
    1
    >>> x[0] = 10
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: 'tuple' object does not support item assignment
    >>>
    ```

However if an immutable object contains a mutable object the mutable object can have its value changed even if it is part of an immutable object. For example, a tuple is an immutable data structure however if a tuple contains a list object, a mutable object, then we can change the value of the list object as shown in the following snippet.

```python
    >>> t = [1, 2, 3, 4]
    >>> x = t,
    >>> x
    ([1, 2, 3, 4],)
    >>> x[0]
    [1, 2, 3, 4]
    >>> x[0].append(10)
    >>> x
    ([1, 2, 3, 4, 10],)
    >>>
```

4.1 *Strong* and *Weak* Object References

Python objects get references when they are bound to names. This binding can be in form of an assignment, a function or method call that binds objects to argument names etc. Every time an object gets a reference, the reference count is increased. In fact the reference count for an object can be found using the `sys.getrefcount` method as shown in the following example.

```
>>> import sys
>>> l = []
>>> m = l
# note that there are 3 references to the list object, l, m and the binding
# to the object argument for sys.getrefcount function
>>> sys.getrefcount(l)
3
```

Two kind of references, **strong and weak references**, exist in Python but when discussing references, it is almost certainly the strong reference that is being referred to. The previous example for instance, has three references and these are all strong references. The defining characteristic of a strong reference in Python is that whenever a new strong reference is created, the reference count for the referenced object is incremented by `1`. This means that the garbage collector will never collect an object that is strongly referenced because the garbage collector collects only objects that have a reference count of `0`. Weak references on the other hand do not increase the reference count of the referenced object. Weak referencing is provided by the `weakref` module. The following snippet shows weak referencing in action.

```
>>> class Foo:
...     pass
...
>>> a = Foo()
>>> b = a
>>> sys.getrefcount(a)
3
>>> c = weakref.ref(a)
>>> sys.getrefcount(a)
3
>>> c()
<__main__.Foo object at 0x1012d6828>
>>> type(c)
<class 'weakref'>
```

The `weakref.ref` function returns an object that when called returns the weakly referenced object. The `weakref` module the `weakref.proxy` alternative to the `weakref.ref` function for creating weak references. This method creates a proxy object that can be used just like the original object without the need for a call as shown in the following snippet.

```
>>> d = weakref.proxy(a)
>>> d
<weakproxy at 0x10138ba98 to Foo at 0x1012d6828>
>>> d.__dict__
{}
```

When all the strong references to an object have deleted then the weak reference looses it reference to the original object and the object is ready for garbage collection. This is shown in the following example.

```
>>> del a
>>> del b
>>> d
<weakproxy at 0x10138ba98 to NoneType at 0x1002040d0>
>>> d.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ReferenceError: weakly-referenced object no longer exists
>>> c()
>>> c().__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute '__dict__'
```

4.2 The Type Hierarchy

Python comes with its own set of built-in types and these built-in types broadly fall into one of the following categories:

`None` Type

The `None` type is a singleton object that has a single value and this value is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned by functions that don't explicitly return a value as illustrated below:

```python
>>> def print_name(name):
...       print(name)
...
>>> name = print_name("nkem")
nkem
>>> name
>>> type(name)
<class 'NoneType'>
>>>
```

The `None` type has a truth value of `false`.

`NotImplemented` Type

The `NotImplemented` type is another singleton object that has a single value. The value of this object is accessed through the built-in name `NotImplemented`. This object should be returned when we want to delegate the search for the implementation of a method to the interpreter rather than throwing a runtime `NotImplementedError` exception. For example, consider the two types, `Foo` and `Bar` below:

```
class Foo:
     def __init__(self, value):
          self.value = value
```

```
    def __eq__(self, other):
        if isinstance(other, Foo):
            print('Comparing an instance of Foo with another instance of Foo')
            return other.value == self.value
        elif isinstance(other, Bar):
            print('Comparing an instance of Foo with an instance of Bar')
            return other.value == self.value
        print('Could not compare an instance of Foo with the other class')
        return NotImplemented

class Bar:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        if isinstance(other, Bar):
            print('Comparing an instance of Bar with another instance of Bar')
            return other.value == self.value
        print('Could not compare an instance of Bar with the other class')
        return NotImplemented
```

When an attempt is made at comparisons, the effect of returning `NotImplemented` can be clearly observed. In Python, `a == b` results in a call to `a.__eq__(b)`. In this example, instance of `Foo` and `Bar` have implementations for comparing themselves to other instance of the same class, for example:

```
>>> f = Foo(1)
>>> b = Bar(1)
>>> f == b
Comparing an instance of Foo with an instance of Bar
True
>>> f == f
Comparing an instance of Foo with another instance of Foo
True
>>> b == b
Comparing an instance of Bar with another instance of Bar
True
>>>
```

What actually happens when we compare `f` with `b`? The implementation of `__eq__()` in `Foo` checks that the `other` argument is an instance of `Bar` and handles it accordingly returning a value of `True`:

```
>>> f == b
Comparing an instance of Foo with an instance of Bar
True
```

If `b` is compared with `f` then `b.__eq__(f)` is invoked and the `NotImplemented` object is returned because the implementation of `__eq__()` in `Bar` only supports comparison with a `Bar` instances. However, it can be seen in the following snippet that the comparison operation actually succeeds; what has happened?

```
>>> b == f
Could not compare an instance of Bar with the other class
Comparing an instance of Foo with an instance of Bar
```

```
True
>>>
```

The call to `b.__eq__(f)` method returned `NotImplemented` causing the python interpreter to invoke the `__eq__()` method in `Foo` and since a comparison between `Foo` and `Bar` is defined in the implementation of the `__eq__()` method in `Foo` the correct result, `True`, is returned.

The `NotImplmented` object has a truth value of true.

```
>>>bool(NotImplemented)
True
```

`Ellipsis` Type

This is another singleton object type that has a single value. The value of this object is accessed through the literal `...` or the built-in name `Ellipsis`. The truth value for the `Ellipsis` object is true. The `Ellipsis` object is mainly used in numeric python for indexing and slicing matrices. The numpy documentation provides more insight into how the `Ellipsis` object is used.

`Numeric Type`

Numeric types are otherwise referred to as numbers. Numeric objects are immutable thus once created their value cannot be changed. Python numbers fall into one of the following categories:

1. Integers: These represent elements from the set of positive and negative integers. These fall into one of the following types:
    1. Plain integers: These are numbers in the range of `-2147483648` through `2147483647` on a 32-bit machine; the range value is dependent on machine word size. Long integers are returned when results of operations fall outside the range of plain integers and in some cases, the exception `OverflowError` is raised. For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user.
    2. Long integers: Long integers are used to hold integer values that are as large as the virtual memory on a system can handle. This is illustrated in the following example.

        ```
        >>> 238**238
        42200323427409150751742179532592018252808661114071266629718376939092568551075505740268077803623642715001998769421215763628719631633783750877563193837256416303318957733860108662430281598286073858990878489423027387093434036402502753142182439305674327314588077348865742839689189553235732976315624152928932760343933360660521328084551181052724703073395502160912535704170505456773718101922384718032634785464920586864837524059460946069784113790792337938047537052436442366076757495221197683115845225278869129420590702227898511756619092052546632633924661341050828869150310 4L
        ```

        It is important to note that from the perspective of a user, there is no difference between the plain and long integers as all conversions if any are done under

covers by the interpreter.

3. Booleans: These represent the truth values `False` and `True`. The `Boolean` type is a subtype of plain integers. The `False` and `True` Boolean values behave like `0` and `1` values respectively except when converted to a string, then the strings "False" or "True" are returned respectively. For example:

```
>>> x = 1
>>> y = True
>>> x + y
2
>>> a = 1
>>> b = False
>>> a + b
1
>>> b == 0
True
>>> y == 1
True
>>>
>>> str(True)
'True'
>>> str(False)
'False'
```

2. Float: These represent machine-level **only** double precision floating point numbers. The underlying machine architecture and specific python implementation determines the accepted range and the handling of overflow; so `CPython` will be limited by the underlying `C` language while `Jython` will be limited by the underlying `Java` language.
3. Complex Numbers: These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. Complex numbers can be created using the `complex` keyword as shown in the following example.

```
>>> complex(1,2)
(1+2j)
>>>
```

Complex numbers can also be created by using a number literal prefixed with a `j`. For instance, the previous complex number example can be created by the expression, `1+2j`. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

Sequence Type

Sequence types are finite ordered collections of objects that can be indexed by integers; using negative indices in python is legal. Sequences fall into two categories - *mutable* and *immutable* sequences.

1. Immutable sequences: An immutable sequence type object is one whose value cannot change once it is created. This means that the collection of objects that are directly referenced by an immutable sequence is fixed. The collection of objects referenced by an immutable sequence maybe composed of mutable objects whose value may

change at runtime but the mutable object itself that is directly referenced by an immutable sequence cannot be changed. For example, a tuple is an immutable sequence but if one of the elements in the tuple is a list, *a mutable sequence*, then the list can change but the reference to the list object that tuple holds cannot be changed as shown below:

```
>>> t = [1, 2, 3], "obi", "ike"
>>> type(t)
<class 'tuple'>
>>> t[0].append(4) # mutate the list
>>> t
([1, 2, 3, 4], 'obi', 'ike')
>>> t[0] = [] # attempt to change the reference in tuple
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The following are built-in immutable sequence types:

1. Strings: A string is an immutable sequence of Unicode code points or more informally an immuta
 sequence of characters. There is no `char` type in python so a character is just a string of le
, 1.
Strings in python can represent all unicode code points in the range `U+0000 - U+10FFFF`. All text
n python is Unicode and the type of the objects used to hold such text is `str`.

2. Bytes: A `bytes` object is an immutable sequence of 8-bit bytes. Each bytes is represented by
integer in the range `0 <= x < 256`. Bytes literals such as `b'abc'` and the built-in function `by
s()` are used to create `bytes` objects. `Bytes` object have an intimate relationship with `string
. Strings are abstractions over text representation used in the computer; text is represented inte
ally using binary or bytes. Strings are just sequences of bytes that have been decoded using an e
ding such as `UTF-8`. The abstract characters of a string can also be encoded using available enc
ngs such as `UTF-8` to get the binary representation of the string in bytes objects. The relation
p between `bytes` and `strings`is illustrated with the following example.

    ```python
    >>> b = b'abc'
    >>> b
    b'abc'
    >>> type(b)
    <class 'bytes'>
    >>> b = bytes('abc', 'utf-16') # encode a string to bytes using UTF-16 encoding
    >>> b
    b'\xff\xfea\x00b\x00c\x00'
    >>> b
    b'\xff\xfea\x00b\x00c\x00'
    >>> b.decode("utf-8") # decoding fails as encoding has been done with utf-16
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte
    >>> b.decode("utf-16") # decoding to string passes
    'abc'
    >>> type(b.decode("utf-16"))
    <class 'str'>
    ```

3. Tuple: A tuple is a sequence of arbitrary python objects. Tuples of two or more items are form
by comma-separated lists of expressions. A tuple of one item is formed by affixing a comma to an

ression while an empty tuple is formed by an empty pair of parentheses. This is illustrated in t following example.

```python
>>> names = "Obi",  # tuple of 1
>>> names
('Obi',)
>>> type(names)
<class 'tuple'>

>>> names = ()  # tuple of 0
>>> names
()
>>> type(names)
<class 'tuple'>


>>> names = "Obi", "Ike", 1  # tuple of 2 or more
>>> names
('Obi', "Ike", 1)
>>> type(names)
<class 'tuple'>
```

1. Mutable sequences: An immutable sequence type is one whose value can change after it has created. There are currently two built-in mutable sequence types - `byte arrays` and `lists`
    1. Byte Arrays: Bytearray objects are mutable arrays of bytes. Byte arrays are created using the built-in `bytearray()` constructor. Apart from being mutable and thus unhashable, byte arrays provide the same interface and functionality as immutable `byte` objects. Bytearrays are very useful when the efficiency offered by their mutability is required. For example, when receiving an unknown amount of data over a network, byte arrays are more efficient because the array can be extended as more data is received without having to allocate new objects as would be the case if the immutable `byte` type was used.
    2. Lists: Lists are a sequence of arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. The empty list is formed with the empty square bracket, `[]`. A list can be created from any iterable by passing such iterable to the `list` method. The list data structure is one of the most widely used data type in python.

Sequence types have some operations that are common to all sequence types. These are described in the following table; *x* is an object, *s* and *t* are sequences and *n*, *i*, *j*, *k* are integers.

Operation	Result
`x in s`	True if an item of s is equal to x, else False
`x not in s`	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n or n * s	n shallow copies of s concatenated
s[i]	ith item of s, origin 0
s[i:j]	slice of s from i to j
s[i:j:k]	slice of s from i to j with step k
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s
s.index(x[, i[, j]])	index of the first occurrence of x in s (at or after index i and before index j)
s.count(x)	total number of occurrences of x in s

Note

1. Values of *n* that are less than 0 are treated as 0 and this yields an empty sequence of the same type as *s* such as below:

```
>>> x = "obi"
>>> x*-2
''
```

2. Copies made from using the `*` operation are shallow copies; any nested structures are not copied. This can result in some confusion when trying to create copies of a structure such as a nested list.

```
>>> lists = [[]] * 3 # shallow copy
>>> lists
[[], [], []] # all three copies reference the same list
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

   To avoid shallow copies when dealing with nested lists, the following method can be adopted

```
```python
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```
```

3. When *i* or *j* is negative, the index is relative to the end of the string thus `len(s) + i` or `len(s) + j` is substituted for the negative value of *i* or *j*.

4. Concatenating immutable sequences such as strings always results in a new object for example:

```
>>> name = "Obi"
>>> id(name)
4330660336
>>> name += "Obi" + " Ike-Nwosu"
>>> id(name)
4330641208
```

Python defines the *interfaces* (thats the closest word that can be used) - `Sequences` and `MutableSequences` in the `collections` library and these define all the methods a *type* must implement to be considered a mutable or immutable sequence; when *abstract base classes* are discussed, this concept will become much clearer.

Set

These are unordered, finite collection of unique python objects. Sets are unordered so they cannot be indexed by integers. The members of a set must be hash-able so only immutable objects can be members of a set. This is so because sets in python are implemented using a hash table; a hash table uses some kind of hash function to compute an index into a slot. If a mutable value is used then the index calculated will change when this object changes thus mutable values are not allowed in sets. Sets provide efficient solutions for membership testing, de-duplication, computing of intersections, union and differences. Sets can be iterated over, and the built-in function `len()` returns the number of items in a set. There are currently two intrinsic set types:- the mutable set type and the immutable frozenset type. Both have a number of common methods that are shown in the following table.

Method	Description
len(s)	return the cardinality of the set, s.
x in s	Test x for membership in s.
x not in s	Test x for non-membership in s.
isdisjoint(other)	Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set.
issubset(other), set <= other	Test whether every element in the set is in other.
set < other	Test whether the set is a proper subset of other, that is, set <= other and set ! other.
issuperset(other), set >= other	Test whether every element in other is in the set.
set > other	Test whether the set is a proper superset of other, that is, set >= other and set != other.
union(other, …), set | other | …	Return a new set with elements from the set and all others.
intersection(other, …), set & other & …	Return a new set with elements common to the set and all others.
difference(other, …), set - other - …	Return a new set with elements in the set that are not in the others.
symmetric_difference(other), set ^ other	Return a new set with elements in either the set or other but not both.
copy()	Return a new set with a shallow copy of s.

1. Frozen set: This represents an immutable set. A frozen set is created by the built-in `frozenset()` constructor. A frozenset is immutable and thus hashable so it can be used as an element of another set, or as a dictionary key.
2. Set: This represents a mutable set and it is created using the built-in `set()` constructor. The mutable set is not hashable and cannot be part of another set. A set can also be created using the set literal `{}`. Methods unique to the mutable set include:

Method	Description
update(other, …), set |= other | …	Update the set, adding elements from all others.
intersection_update(other, …), set &= other & …	Update the set, keeping only elements found in it and all others.
difference_update(other, …), set -= other | …	Update the set, removing elements found in others.
symmetric_difference_update(other), set ^= other	Update the set, keeping only elements found in either set, but not in both.
add(elem)	Add element elem to the set.
remove(elem)	Remove element elem from the set. Raises KeyError if elem is not contained in the set.
discard(elem)	Remove element elem from the set if it is present.
pop()	Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.
clear()	Remove all elements from the set.

Mapping

A python mapping is a finite set of objects (values) indexed by a set of immutable python objects (keys). The keys in the mapping must be *hashable* for the same reason given previously in describing set members thus eliminating mutable types like lists, frozensets, mappings etc. The expression, `a[k]`, selects the item indexed by the key, `k`, from the mapping `a` and can be used as in assignments or `del` statements. The `dictionary` mostly called `dict` for convenience is the only intrinsic mapping type built into python:

1. Dictionary: Dictionaries can be created by placing a comma-separated sequence of `key: value` pairs within braces, for example: `{'name': "obi", 'age': 18}`, or by the *dict()* constructor. The main operations supported by the `dictionary` type is the addition, deletion and selection of values using a given key. When adding a key that is already in use within a dict, the old value associated with that key is forgotten. Attempting to access a value with a non-existent key will result in a `KeyError` exception. Dictionaries are perhaps one of the most important types within the interpreter. Without explicitly making use of a dictionary, the interpreter is already using them in a number of different places. For example, the namespaces, namespaces are discussed in a subsequent chapter, in python are implemented using dictionaries; this means that every time a symbol is referenced within a program, a dictionary access occurs. Objects are layered on dictionaries in python; all attributes of python objects are stored in a dictionary attribute, `__dict__`. These are but a few applications of this type within the python interpreter.

Python supplies more advanced forms of the dictionary type in its `collections` library. These are the `OrderedDict` that introduces order into a dictionary thus remembering the order in which items were insert and the `defaultdict` that takes a factory function that is called to produce a value when a key is missing. If a key is missing from a defaultdict instance, the factory function is called to produce a value for the key and the dictionary is updated with this key, value pair and the created value is returned. For example,

```python
    >>> from collections import defaultdict
    >>> d = defaultdict(int)
    >>> d
    defaultdict(<class 'int'>, {})
    >>> d[7]
    0
    >>>d
    defaultdict(<class 'int'>, {7: 0})
```

Callable Types

These are types that support the function call operation. The function call operation is the use of `()` after the type name. In the example below, the function is `print_name` and the function call is when the `()` is appended to the function name as such `print_name()`.

```
    def print_name(name):
        print(name)
```

Functions are not the only callable types in python; any object type that implements the `__call__` special method is a callable type. The function, `callable(type)`, is used to check that a given type is *callable*. The following are built-in callable python types:

1. User-defined functions: these are functions that a user defines with the `def` statement such as the `print_name` function from the previous section.
2. Methods: these are functions defined within a class and accessible within the scope of the class or a class instance. These methods could either be instance methods, static or class methods.
3. Built-in functions: These are functions available within the interpreter core such as the `len` function.
4. Classes: Classes are also callable types. The process of creating a class instance involves calling the class such as `Foo()`.

Each of the above types is covered in detail in subsequent chapters.

Custom Type

Custom types are created using the `class` statements. Custom class objects have a type of `type`. These are types created by user defined programs and they are discussed in the chapter on object oriented programming.

Module Type

A module is one of the organizational units of Python code just like functions or classes. A

module is also an object just like every other value in the python. The module type is created by the import system as invoked either by the import statement, or by calling functions such as `importlib.import_module()` and built-in `__import__()`.

File/IO Types

A file object represents an open file. Files are created using the open built-in functions that opens and returns a file object on the local file system; the file object can be open in either binary or text mode. Other methods for creating file objects include:

1. `os.fdopen` that takes a file descriptor and create a file object from it. The `os.open` method not to be confused with the `open` built-in function is used to create a file descriptor that can then be passed to the `os.fdopen` method to create a file object as shown in the following example.

```
>>> import os
>> fd = os.open("test.txt", os.O_RDWR|os.O_CREAT)
>>> type(fd)
<class 'int'>
>>> fd
3
>>> fo = os.fdopen(fd, "w")
>>> fo
<_io.TextIOWrapper name=3 mode='w' encoding='UTF-8'>
>>> type(fo)
<class '_io.TextIOWrapper'>
```

2. `os.popen()`: this is marked for deprecation.
3. `makefile()` method of a socket object that opens and returns a file object that is associated with the socket on which it was called.

The built-in objects, `sys.stdin`, `sys.stdout` and `sys.stderr`, are also file objects corresponding to the python interpreter's standard input, output and error streams.

Built-in Types

These are objects used internally by the python interpreter but accessible by a user program. They include traceback objects, code objects, frame objects and slice objects

Code Objects

Code objects represent compiled executable Python code, or bytecode. Code objects are machine code for the python virtual machine along with all that is necessary for the execution of the bytecode they represent. They are normally created when a *block of code* is compiled. This executable piece of code can only be executed using the `exec` or `eval` python methods. To give a concrete understanding of code objects we define a very simple function below and dissect the code object.

```
def return_author_name():
    return "obi Ike-Nwosu"
```

The code object for the above function can be obtained from the function object by assessing its `__code__` attribute as shown below:

```
>>> return_author_name.__code__
<code object return_author_name at 0x102279270, file "<stdin>", line 1>
```

We can go further and inspect the code object using the `dir` function to see the attributes of the code object.

```
 >>> dir(return_author_name.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr
bute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '
_reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'co_argcount'
, 'co_cellvars', 'co_code', 'co_consts', 'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars',
 'co_kwonlyargcount', 'co_lnotab', 'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames
']
```

Of particular interest to us at this point in time are the non-special methods that is methods that do not start with an underscore. We give a brief description of each of these non-special methods in the following table

Method	Description
co_argcount	number of arguments (not including * or ** args)
co_code	string of raw compiled bytecode
co_consts	tuple of constants used in the bytecode
co_filename	name of file in which this code object was created
co_firstlineno	number of first line in Python source code
co_flags	bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
co_lnotab	encoded mapping of line numbers to bytecode indices
co_name	name with which this code object was defined
co_names	tuple of names of local variables
co_nlocals	number of local variables
co_stacksize	virtual machine stack space required
co_varnames	tuple of names of arguments and local variables

We can view the bytecode string for the function using the `co_code` method of the code object as shown below.

```
>>> return_author_name.__code__.co_code
b'd\x01\x00S'
```

The bytecode returned however is basically of no use to someone investigating code objects. This is where the python `dis` module comes into play. The `dis` module can be used to generate a human readable version of the code object. We use the `dis` function from the `dis` module to generate the code object for `return_author_name` function.

```
>>> dis.dis(return_author_name)
  2           0 LOAD_CONST               1 ('obi Ike-Nwosu')
              3 RETURN_VALUE
```

The above shows the human readable version of the the python code object. The `LOAD_CONST` instruction reads a value from the `co_consts` tuple, and pushes it onto the top of the stack (the *CPython* interpreter is a stack based virtual machine). The `RETURN_VALUE` instruction pops the top of the stack, and returns this to the calling scope signalling the end of the execution of that python code block.

Code objects serve a number of purposes while programming. They contain information that can aid in interactive debugging while programming and can provide us with readable tracebacks during an exception.

Frame Objects

Frame objects represent execution frames. Python code blocks are executed in execution frames. The call stack of the interpreter stores information about currently executing subroutines and the call stack is made up of stack **frame objects**. Frame objects on the stack have a *one-to-one* mapping with subroutine calls by the program executing or the interpreter. The frame object contains code objects and all necessary information, including references to the local and global name-spaces, necessary for the runtime execution environment. The frame objects are linked together to form the call stack. To simplify how this all fits together a bit, the call stack can be thought of as a stack data structure (it actually is), every time a subroutine is called, a frame object is created and inserted into the stack and then the code object contained within the frame is executed. Some special read-only attributes of frame objects include:

1. `f_back` is to the previous stack frame towards the caller, or `None` if this is the bottom stack frame.
2. `f_code` is the code object being executed in this frame.
3. `f_locals` is the dictionary used to look up local variables.
4. `f_globals` is used for global variables.
5. `f_builtins` is used for built-in names.
6. `f_lasti` gives the precise instruction - it is an index into the bytecode string of the code object.

Some special writable attributes include:

1. `f_trace`: If this is not `None`, this is a function called at the start of each source code line.
2. `f_lineno`: This is the current line number of the frame. Writing to this from within a trace function jumps to the given line only for the bottom-most frame. A debugger can implement a `Jump` command by writing to `f_lineno`.

Frame objects support one method:

1. `frame.clear()`: This method clears all references to local variables held by the frame. If the frame belonged to a generator, the generator is finalized. This helps break reference cycles involving frame objects. A `RuntimeError` is raised if the frame is currently executing.

Traceback Objects

Traceback objects represent the stack trace of an exception. A traceback object is created when an exception occurs. The interpreter searches for an exception handler by continuously popping the execution stack and inserting a traceback object in front of the current traceback for each frame popped. When an exception handler is encountered, the stack trace is made available to the program. The stack trace object is accessible as the third item of the tuple returned by `sys.exc_info()`. When the program contains no suitable handler, the stack trace is written to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`. A few important attributes of a traceback object is shown in the following table.

Method	Description
tb_next	is the next level in the stack trace (towards the frame where the exception occurred), or None if there is no next level
tb_frame	points to the execution frame of the current level; tb_lineno gives the line number where the exception occurred
tb_lasti	indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a try statement with no matching except clause or with a finally clause.

Slice Objects

Slice objects represent slices for `__getitem__()` methods of sequence-like objects (more on special methods such as `__getitem__()` in the chapter on object oriented programming). Slice object return a subset of the sequence they are applied to as shown below.

```
>>> t = [i for i in range(10)]
>>> t
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t[:10:2]
[0, 2, 4, 6, 8]
>>>
```

They are also created by the built-in `slice([start,], stop [,step])` function. The returned object can be used in between the square brackets as a regular slice object.

```
>>> t = [i for i in range(10)]
>>> t
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t[:10:2]
[0, 2, 4, 6, 8]
>>> s = slice(None, 10, 2)
>>> s
slice(None, 10, 2)
>>> t[s]
[0, 2, 4, 6, 8]
```

Slice object read-only attributes include:

Attribute	Description
`start`	which is the lower bound;
`stop`	the optional upper bound;
`step`	the optional step value;

Each of the optional attributes is `None` if omitted. Slices can take a number of forms in addition to the standard `slice(start, stop [,step])`. Other forms include

```
a[start:end] # items start to end-1 equivalent to slice(start, stop)
a[start:]     # items start to end-1 equivalent to slice(start)
a[:end]       # items from the beginning to end-1 equivalent to slice(None, end)
a[:]          # a shallow copy of the whole array equivalent to slice(None, None)
```

The start or end values may also be negative in which case we count from the end of the array as shown below:

```
a[-1]     # last item in the array equivalent to slice(-1)
a[-2:]    # last two items in the array equivalent to slice(-2)
a[:-2]    # everything except the last two items equivalent to slice(None, -2)
```

Slice objects support one method:

1. slice.indices(self, length): This method takes a single integer argument, length, and returns a tuple of three integers - (start, stop, stride) that indicates how the slice would apply to the given length. The start and stop indices are actual indices they would be in a sequence of length given by the length argument. An example is shown below:

```python
>>> s = slice(10, 30, 1)
# applying slice(10, 30, 1) to sequence of length 100 gives [10:30]
>>> s.indices(100)
(10, 30, 1)
# applying slice(10, 30, 1) to sequence of length 15 gives [10:15]
>>> s.indices(15)
(10, 15, 1)
# applying slice(10, 30, 1) to sequence of length 1 gives [1:1]
>>> s.indices(1)
(1, 1, 1)
>>> s.indices(0)
(0, 0, 1)
```

Generator Objects

Generator objects are created by the invocation of generator functions; these are functions that use the keyword, `yield`. This type is discussed in detail in the chapter on `Sequences and Generators`.

With a strong understanding of the built-in type hierarchy, the stage is now set for examining object oriented programming and how users can create their own type hierarchy and even make such types behave like built-in types.

5. Object Oriented Programming

Classes are the basis of object oriented programming in python and are one of the basic organizational units in a python program.

5.1 The Mechanics of Class Definitions

The `class` statement is used to define a new type. The class statement defines a set of attributes, variables and methods, that are associated with and shared by a collection of instances of such a class. A simple class definition is given below:

```
class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt

     def inquiry(self):
          return self.balance
```

Class definitions introduce class objects, instance objects and method objects.

Class Objects

The execution of a class statement creates a class object. At the start of the execution of a class statement, a new *name-space* is created and this serves as the name-space into which all class attributes go; unlike languages like Java, this name-space does not create a new local scope that can be used by class methods hence the need for fully qualified names when accessing attributes. The `Account` class from the previous section illustrates this; a method trying to access the `num_accounts` variable must use the fully qualified name, `Account.num_accounts` else an error results such as when the fully qualified name is not used in the `__init__` method as shown below:

```
class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
```

```
            self.balance = balance
            num_accounts += 1

        def del_account(self):
            Account.num_accounts -= 1

        def deposit(self, amt):
            self.balance = self.balance + amt

        def withdraw(self, amt):
            self.balance = self.balance - amt

        def inquiry(self):
            return self.balance
```

```
>>> acct = Account('obi', 10)
Traceback (most recent call last):
  File "python", line 1, in <module>
  File "python", line 9, in __init__
UnboundLocalError: local variable 'num_accounts' referenced before assignment
```

At the end of the execution of a class statement, a class object is created; the scope preceding the class definition is reinstated, and the class object is bound in this scope to the class name given in the class definition header.

A little diversion here. One may ask, *if the class created is an object then what is the class of the class object?*. In accordance with the Python philosophy that *every value is an object* , the class object does indeed have a class which it is created from; this is the `type` class.

```
>>> type(Account)
<class 'type'>
```

So just to confuse you a bit, the type of a type, *the Account type*, is `type`. To get a better understanding of the fact that a class is indeed an object with its own class we go behind the scenes to explain what really goes on during the execution of a class statement using the `Account` example from above.

```
>>>class_name = "Account"
>>>class_parents = (object,)
>>>class_body = """
num_accounts = 0

def __init__(self, name, balance):
    self.name = name
    self.balance = balance
    num_accounts += 1

def del_account(self):
    Account.num_accounts -= 1

def deposit(self, amt):
    self.balance = self.balance + amt

def withdraw(self, amt):
    self.balance = self.balance - amt
```

```
    def inquiry(self):
        return self.balance
    """
     # a new dict is used as local name-space
     >>>class_dict = {}

     #the body of the class is executed using dict from above as local
     # name-space
     >>>exec(class_body, globals(), class_dict)

     # viewing the class dict reveals the name bindings from class body
     >>> class_dict
     {'del_account': <function del_account at 0x106be60c8>, 'num_accounts': 0, 'inquiry': <function
nquiry at 0x106beac80>, 'deposit': <function deposit at 0x106be66e0>, 'withdraw': <function withdra
 at 0x106be6de8>, '__init__': <function __init__ at 0x106be2c08>}

     # final step of class creation
     >>>Foo = type(class_name, class_parents, class_dict)
     # view created class object
     >>>Account
     <class '__main__.Account'>
     >>>type(Account)
     <type 'type'>
```

During the execution of class statement, the interpreter carries out the following steps behind the scene:

1. The body of the class statement is isolated in a string.
2. A class dictionary representing the name-space for the class is created.
3. The body of the class is executed as a set of statements within this name-space.
4. During the final step, the class object is created by instantiating the `type` class, passing in the class name, base classes and class dictionary as arguments. The `type` class used here in creating the `Account` class object is a **meta-class**, the class of a class. The meta-class used in the class object creation can be explicitly specified by supplying the `metaclass` keyword argument in the `class` definition. In the case that this is not supplied, the class statement examines the first entry in the *tuple* of the the base classes if any. If no base classes are used, the global variable `__metaclass__` is searched for and if no value is found for this, Python uses the default meta-class. More about meta-classes is discussed in subsequent chapters.

Class objects support *attribute reference* and *object instantiation*. Attributes are referenced using the standard dot syntax; an object followed by dot and then attribute name: `obj.name`. Valid attribute names are all the variable and method names present in the class' name-space when the class object was created. For example:

```
    >>> Account.num_accounts
    0
    >>> Account.deposit
    >>> <unbound method Account.deposit>
```

Object instantiation is carried out by calling the class object like a normal function with required parameters for the `__init__` method of the class as shown in the following

example:

```
>>> Account("obi", 0)
```

An instance object that has been initialized with supplied arguments is returned from instantiation of a class object. In the case of the `Account` class, the account name and account balance are set and, the number of instances is incremented by `1` in the `__init__` method.

Instance Objects

If class objects are the cookie cutters then instance objects are the cookies that are the result of instantiating class objects. Instance objects are returned after the correct initialization of a class just as shown in the previous section. Attribute references are the only operations that are valid on instance objects. Instance attributes are either data attribute, better known as instance variables in languages like Java, or method attributes.

Method Objects

If `x` is an instance of the `Account` class, `x.deposit` is an example of a method object. Method objects are similar to functions however during a method definition, an extra argument is included in the arguments list, the `self` argument. This `self` argument refers to an instance of the class but *why do we have to pass an instance as an argument to a method?* This is best illustrated by a method call such as the following.

```
>>> x = Account()
>>> x.inquiry()
10
```

But what exactly happens when an instance method is called? It can be observed that `x.inquiry()` is called without an argument above, even though the method definition for `inquiry()` requires the `self` argument. *What happened to this argument?*

In the example from above, the call to `x.inquiry()` is exactly equivalent to `Account.inquiry(x)`; notice that the object instance, `x`, is being passed as argument to the method - this is the `self` argument. Invoking an object method with an argument list is equivalent to invoking the corresponding method from the object's class with an argument list that is created by inserting the method's object at the start of the list of argument. In order to understand this, note that methods are stored as functions in class `dicts`.

```
>>> type(Account.inquiry)
<class 'function'>
```

To fully understand how this transformation takes place one has to understand descriptors and Python's attribute references algorithm. These are discussed in subsequent sections of this chapter. In summary, the method object is a wrapper around a function object; when the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the underlying function object is called with this new argument list. This applies to all instance method objects including the `__init__` method. Note that the `self` argument is actually not a keyword; the name, `self` is just a

convention and any valid argument name can be used as shown in the `Account` class definition below.

```
class Account(object):
     num_accounts = 0

     def __init__(obj, name, balance):
          obj.name = name
          obj.balance = balance
          Account.num_accounts += 1

     def del_account(obj):
          Account.num_accounts -= 1

     def deposit(obj, amt):
          obj.balance = obj.balance + amt

     def withdraw(obj, amt):
          obj.balance = obj.balance - amt

     def inquiry(obj):
          return obj.balance
```

```
>>> Account.num_accounts
0
>>> x = Account('obi', 0)
>>> x.deposit(10)
>>> Account.inquiry(x)
10
```

5.2 Customizing User-defined Types

Python is a very flexible language providing user with the ability to customize classes in ways that are unimaginable in other languages. Attribute access, class creation and object initialization are a few examples of ways in which classes can be customized. User defined types can also be customized to behave like built-in types and support special operators and syntax such as `*`, `+`, `-`, `[]` etc.

All these customization is possible because of methods that are called *special* or *magic* methods. Python special methods are just ordinary python methods with double underscores as prefix and suffix to the method names. Special methods have already encountered in this book. An example is the `__init__` method that is called to initialize class instances; another is the `__getitem__` method invoked by the index, `[]` operator; an index such as `a[i]` is translated by the interpreter to a call to `type(a).__getitem__(a, i)`. Methods with the double underscore as prefix and suffix are just ordinary python methods; users can define their own class methods with method names prefixed and suffixed with the double underscore and use it just like normal python methods. This is however not the conventional approach to defining normal user methods.

User defined classes can also implement these special methods; a corollary of this is that built-in operators such as `+` or `[]` can be adapted for use by user defined classes. This is one of the essence of *polymorphism* in Python. In this book, special methods are grouped according to the functions they serve. These groups include:

Special methods for instance creation

The __new__ and __init__ special methods are the two methods that are integral to instance creation. New class instances are created in a two step process; first the static method, __new__, is called to create and return a new class instance then the __init__ method is called to to initialize the newly created object with supplied arguments. A very important instance in which there is a need to override the __new__ method is when sub-classing built-in immutable types. Any initialization that is done in the sub-class must be done before object creation. This is because once an immutable object is created, its value cannot be changed so it makes no sense trying to carry out any function that modifies the created object in an __init__ method. An example of sub-classing is shown in the following snippet in which whatever value is supplied is rounded up to the next integer.

```
>>> import math
>>> class NextInteger(int):
...       def __new__(cls, val):
...            return int.__new__(cls, math.ceil(val))
...
>>> NextInteger(2.2)
3
>>>
```

Attempting to do the `math.ceil` operation in an __init__ method will cause the object initialization to fail. The __new__ method can also be overridden to create a `Singleton` super class; subclasses of this class can only ever have a single instance throughout the execution of a program; the following example illustrates this.

```
class Singleton:

     def __new__(cls, *args, **kwds):
          it = cls.__dict__.get("__it__")
          if it is None:
               return it
          cls.__it__ = it = object.__new__(cls)
          it.init(*args, **kwds)
          return it

     def __init__(self, *args, **kwsds):
          pass
```

It is worth noting that when implementing the __new__ method, the implementation must call its base class' __new__ and the implementation method must return an object.

Users are already familiar with defining the __init__ method; the __init__ method is overridden to perform attribute initialization for an instance of a mutable types.

Special methods for attribute access

The special methods in this category provide means for customizing attribute references; this maybe in order to access or set such an attribute. This set of special methods available for this include:

1. __getattr__: This method can be implemented to handle situations in which a referenced attribute cannot be found. This method is ***only*** called when an attribute

that is referenced is neither an instance attribute nor is it found in the class tree of that object. This method should return some value for the attribute or raise an `AttributeError` exception. For example, if *x* is an instance of the `Account` class defined above, trying to access an attribute that does not exist will result in a call to this method as shown in the following snippet

```
class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def __getattr__(self, name):
          return "Hey I don't see any attribute called {}".format(name)

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt

     def inquiry(self):
          return "Name={}, balance={}".format(self.name, self.balance)

>>> x = Account('obi', 0)
>>> x.balaance
Hey I dont see any attribute called balaance
```

Care should be taken with the implementation of `__getattr__` because if the implementation references an instance attribute that does not exist, an infinite loop may occur because the `__getattr__` method is called successively without end.

```
class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def __getattr__(self, name):
          return self.namee # trying to acess a variable that doesnt exist will result in __geta
r___ calling itself over and over again

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt
```

```
        def inquiry(self):
             return "Name={}, balance={}".format(self.name, self.balance)

    >>> x = Account('obi', 0)
    >>> x.balaance # this will result in a RuntimeError: maximum recursion depth exceeded while call
ing a Python object exception
```

1. `__getattribute__`: This method is implemented to customize the attribute access for a class. This method is **always** called unconditionally during attribute access for instances of a class.
2. `__setattr__`: This method is implemented to unconditionally handle all attribute assignment. `__setattr__` should insert the value being assigned into the dictionary of the instance attributes rather than using `self.name=value` which results in an infinite recursive call. When `__setattr__()` is used for instance attribute assignment, the base class method with the same name should be called such as `super().__setattr__(self, name, value)`.
3. `__delattr__`: This is implemented to customize the process of deleting an instance of a class. it is invoked whenever `del obj` is called.
4. `__dir__`: This is implemented to customize the list of object attributes returned by a call to `dir(obj)`.

Special methods for Type Emulation

Built-in types in python have special operators that work with them. For example, numeric types support the `+` operator for adding two numbers, numeric types also support the `-` operator for subtracting two numbers, sequence and mapping types support the `[]` operator for indexing values held. Sequence types even also have support for the `+` operator for concatenating such sequences. User defined classes can be customized to *behave* like these built-in types where it makes sense. This can be done by implementing the special methods that are invoked by the interpreter when these *special* operators are encountered. The special methods that provide these functionalities for emulating built-in types can be broadly grouped into one of the following:

Numeric Type Special Methods

The following table shows some of the basic operators and the special methods invoked when these operators are encountered.

Special Method	Operator Description
`a.__add__(self, b)`	binary addition, a + b
`a.__sub__(self, b)`	binary subtraction, a - b
`a.__mul__(self, b)`	binary multiplication, a * b
`a.__truediv__(self, b)`	division of a by b
`a.__floordiv__(self, b)`	truncating division of a by b
`a.__mod__(self, b)`	a modulo b
`a.__divmod__(self, b)`	returns a divided by b, a modulo b
`a.__pow__(self, b[, modulo])`	a raised to the b*th* power

Python has the concept of reflected operations; this was covered in the section on the `NotImplemented` of previous chapter. The idea behind this concept is that if the left operand of a binary arithmetic operation does not support a required operation and returns `NotImplemented` then an attempt is made to call the corresponding reflected operation on the right operand provided the type of both operands differ. An example of this rarely used functionality is shown in the following trivial example for emphasis.

```
class MyNumber(object):
     def __init__(self, x):
          self.x = x

     def __str__(self):
          return str(self.x)

>>> 10 - MyNumber(9) # int type, 10, does not know how to subtract MyNumber type and MyNum
r does not know how to handle the operation too
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'MyNumber'
```

In the next snippet the class implements the reflected special method and this reflected method is called by the interpreter.

```
class MyFixedNumber(MyNumber):
     def __rsub__(self, other): # reflected operation implemented
          return MyNumber(other - self.val)

>>> (10 - MyFixedNumber(9)).val
1
```

The following special methods implement reflected binary arithmetic operations.

Special Method	Operator Description
`a.__radd__(self, b)`	reflected binary addition, a + b
`a.__rsub__(self, b)`	reflected binary subtraction, a - b
`a.__rmul__(self, b)`	reflected binary multiplication, a * b
`a.__rtruediv__(self, b)`	reflected division of a by b
`a.__rfloordiv__(self, b)`	reflected truncating division of a by b
`a.__rmod__(self, b)`	reflected a modulo b
`a.__rdivmod__(self, b)`	reflected a divided by b, a modulo b
`a.__rpow__(self, b[, modulo])`	reflected a raised to the b*th* power

Another set of operators that work with numeric types are the augmented assignment operators. An example of an augmented operation is shown in the following code snippet.

```
>>> val = 10
>>> val += 90
>>> val
```

```
100
>>>
```

A few of the special methods for implementing augmented arithmetic operations are listed in the following table.

Special Method	Description
`a.__iadd__(self, b)`	a += b
`a.__isub__(self, b)`	a -= b
`a.__imul__(self, b)`	a *= b
`a.__itruediv__(self, b)`	a //= b
`a.__ifloordiv__(self, b)`	a /= b
`a.__imod__(self, b)`	a %= b
`a.__ipow__(self, b[, modulo])`	a **= b

Sequence and Mapping Types Special Methods

Sequence and mapping are often referred to as container types because they can hold references to other objects. User-defined classes can emulate container types to the extent that this makes sense if such classes implement the special methods listed in the following table.

Special Method	Description
`__len__(obj)`	returns length of `obj`. This is invoked to implement the built-in function `len()`. An object that doesn't define a `__bool__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context.
`__getitem__(obj, key)`	fetches item, `obj[key]`. For sequence types, the keys should be integers or slice objects. If key is of an inappropriate type, `TypeError` may be raised; if the key has a value outside the set of indices for the sequence, `IndexError` should be raised. For mapping types, if key is absent from the container, `KeyError` should be raised.
`__setitem__(obj, key, value)`	Sets `obj[key] = value`
`__delitem__(obj, key)`	deletes `obj[key]`. Invoked by del `obj[key]`
`__contains__(obj, key)`	Returns true if `key` is contained in `obj` and false otherwise. Invoked by a call to `key in obj`
`__iter__(self)`	This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container. Iterator objects also need to implement this method; they are required to return themselves. This is also used by the `for..in` construct.

Sequence types such as lists support the addition (for concatenating lists) and multiplication operators (for creating copies), `+ and *` respectively, by defining the methods `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` and `__imul__()`. Sequence types also implement the `__reversed__` method that implements the `reversed()` method that is used for reverse iteration over a sequence. User defined classes can implement these special methods to get the required functionality.

Emulating Callable Types

Callable types support the function call syntax, `(args)`. Classes that implement the `__call__(self[, args…])` method are callable. User defined classes for which this functionality makes sense can implement this method to make class instances callable. The following example shows a class implementing the `__call__(self[, args…])` method and how instances of this class can be called using the function call syntax.

```
class Account(object):
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
```

```
            Account.num_accounts += 1

        def __call__(self, arg):
            return "I was called with \'{}\'".format(arg)

        def del_account(self):
            Account.num_accounts -= 1

        def deposit(self, amt):
            self.balance = self.balance + amt

        def withdraw(self, amt):
            self.balance = self.balance - amt

        def inquiry(self):
            return self.balance

>>> acct = Account()
>>> acct("Testing function call on instance object")
I was called with 'Testing function call on instance object'
```

Special Methods for comparing objects

User-defined classes can provide custom implementation for the special methods invoked by the five object comparison operators in python, <, >, >=, <=, = in order to control how these operators work. These special methods are given in the following table.

Special Method	Description
`a.__lt__(self, b)`	a < b
`a.__le__(self, b)`	a <= b
`a.__eq__(self, b)`	a == b
`a.__ne__(self, b)`	a != b
`a.__gt__(self, b)`	a > b
`a.__ge__(self, b)`	a >= b

In Python, `x==y` is `True` does not imply that `x!=y` is `False` so `__eq__()` should be defined along with `__ne__()` so that the operators are well behaved. `__lt__()` and `__gt__()`, and `__le__()` and `__ge__()` are each other's reflection while `__eq__()` and `__ne__()` are their own reflection; this means that if a call to the implementation of any of these methods on the left argument returns `NotImplemented`, the reflected operator is is used.

Special Methods and Attributes for Miscellaneous Customizations

1. `__slots__`: This is a special attribute rather than a method. It is an optimization trick that is used by the interpreter to efficiently store object attributes. Objects by default store all attributes in a dictionary (the `__dict__` attribute) and this is very inefficient when objects with few attributes are created in large numbers. `__slots__` make use of a static iterable that reserves just enough space for each attribute rather than the dynamic `__dict__` attribute. The iterable representing the `__slot__` variable can also be a string made up of the attribute names. The following example shows how `__slots__` works.

```
class Account:
     """base class for representing user accounts"""

     # we can also use __slots__ = "name balance"
     __slots__ = ['name', 'balance']
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def __getattr__(self, name):
          """handle attribute reference for non-existent attribute"""
          return "Hey I dont see any attribute called {}".format(name)

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt

     def inquiry(self):
          return "Name={}, balance={}".format(self.name, self.balance)

>>>acct = Account("obi", 10)
>>>acct.__dict__ # __dict__ attribute is gone
Hey I dont see any attribute called __dict__
>>>acct.x = 10
Traceback (most recent call last):
File "acct.py", line 32, in <module>
  acct.x = 10
AttributeError: 'Account' object has no attribute 'x'
>>>acct.__slots__
['name', 'balance']
```

A few things that are worth noting about `__slots__` include the following:

1. If a superclass has the `__dict__` attribute then using `__slots__` in sub-classes is of no use as the dictionary is available.
2. If `__slots__` are used then attempting to assign to a variable not in the `__slots__` variable will result in an `AttributeError` as shown in the previous example.
3. Sub-classes will have a `__dict__` even if they inherit from a base class with a `__slots__` declaration; subclasses have to define their own `__slots__` attribute which must contain only the additional names in order to avoid having the `__dict__` for storing names.
4. Subclasses with “variable-length” built-in types as base class cannot have a non-empty `__slots__` variable.
5. `__bool__`: This method implements the truth value testing for a given class; it is invoked by the built-in operation `bool()` and should return a `True` or `False` value. In the absence of an implementation, `__len__()` is called and if `__len__` is

implemented, the object's truth value is considered to be `True` if result of the call to `__len__` is non-zero. If neither `__len__()` nor `__bool__()` are defined by a class then all its instances are considered to be `True`.

6. `__repr__` and `__str__`: These are two closely related methods as they both return string representations for a given object and only differ subtly in the intent behind their creation. Both are invoked by a call to `repr` and `str` methods respectively. The `__repr__` method implementation should return an unambiguous string representation of the object it is being called on. **Ideally**, the representation that is returned should be an expression that when evaluated by the `eval` method returns the given object; when this is not possible the representation returned should be as unambiguous as possible. On the other hand, `__str__` exists to provide a human readable version of an object; a version that would make sense to some one reading the output but that doesn't necessarily understand the semantics of the language. A very good illustration of how both methods differ is shown below by calling both methods on a data object.

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str(today)
'2015-07-05 20:55:58.642018' # human readable version of datetime object
>>> repr(today)
'datetime.datetime(2015, 7, 5, 20, 55, 58, 642018)' # eval will return the datetime objec
```

   When using string interpolation, `%r` makes a call to `repr` while `%s` makes a call to `str`.

7. `__bytes__`: This is invoked by a call to the `bytes()` built-in and it should return a byte string representation for an object. The byte string should be a `bytes` object.
8. `__hash__`: This is invoked by the `hash()` built-in. It is also used by operations that work on types such as `set`, `frozenset`, and `dict` that make use of object hash values. Providing `__hash__` implementation for user defined classes is an involved and delicate act that should be carried out with care as will be seen. Immutable built-in types are hashable while mutable types such as lists are not. For example, the hash of a number is the value of the number as shown in the following snippet.

```
>>> hash(1)
1
>>> hash(12345)
12345
>>>
```

User defined classes have a default hash value that is derived from their `id()` value. Any `__hash__()` implementation must return an integer and objects that are equal by comparison must have the same hash value so for two object, `a` and `b`, `(a==b and hash(a)==hash(b))` must be true. A few rules for implementing a `__hash__()` method include the following: 1. A class should only define the `__hash__()` method if it also defines the `__eq__()` method.

1. The absence of an implementation for the `__hash__()` method in a class renders its instances unhashable.

2. The interpreter provides user-defined classes with default implementations for `__eq__()` and `__hash__()`. By default, all objects compare unequal except with themselves and `x.__hash__()` returns a value such that (`x == y` and `x is y` and `hash(x) == hash(y)`) is always true. In *CPython*, the default `__hash__()` implementation returns a value derived from the `id()` of the object.
3. Overriding the `__eq__()` method without defining the `__hash__()` method sets the `__hash__()` method to `None` in the class. When the `__hash__()` method of a class is `None`, an instance of the class will raise an appropriate `TypeError` when an attempt is made to retrieve its hash value. The object will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)`.
4. If a class overrides the `__eq__()` and needs to keep the implementation of `__hash__()` from a base class, this must be done explicitly by setting `__hash__ = BaseClass.__hash__`.
5. A class that does not override the `__eq__()` can suppress hash support by setting `__hash__` to `None`. If a class defines its own `__hash__()` method that explicitly raises a `TypeError`, instances of such class will be incorrectly identified as hashable by an `isinstance(obj, collections.Hashable)` test.

5.3 A `Vector` class

In this section, a complete example of the use of special methods to emulate built-in types is provided by a `Vector` class. The `Vector` class provides support for performing vector arithmetic operations.

```
# Copyright 2013 Philip N. Klein
class Vec:
     """
  A vector has two fields:
  D - the domain (a set)
  f - a dictionary mapping (some) domain elements to field elements
      elements of D not appearing in f are implicitly mapped to zero
  """
     def __init__(self, labels, function):
          assert isinstance(labels, set)
          assert isinstance(function, dict)
          self.D = labels
          self.f = function


     def __getitem__(self, key):
          """
      Return the value of entry k in v.
      Be sure getitem(v,k) returns 0 if k is not represented in v.f.

      >>> v = Vec({'a','b','c', 'd'},{'a':2,'c':1,'d':3})
      >>> v['d']
      3
      >>> v['b']
      0
      """
          assert key in self.D
          if key in self.f:
               return self.f[key]
```

```
        return 0

    def __setitem__(self, key, val):
        """
    Set the element of v with label d to be val.
    setitem(v,d,val) should set the value for key d even if d
    is not previously represented in v.f, and even if val is 0.

    >>> v = Vec({'a', 'b', 'c'}, {'b':0})
    >>> v['b'] = 5
    >>> v['b']
    5
    >>> v['a'] = 1
    >>> v['a']
    1
    >>> v['a'] = 0
    >>> v['a']
    0
    """
        assert key in self.D
        self.f[key] = val

    def __neg__(self):
        """
    Returns the negation of a vector.

    >>> u = Vec({1,3,5,7},{1:1,3:2,5:3,7:4})
    >>> -u
    Vec({1, 3, 5, 7},{1: -1, 3: -2, 5: -3, 7: -4})
    >>> u == Vec({1,3,5,7},{1:1,3:2,5:3,7:4})
    True
    >>> -Vec({'a','b','c'}, {'a':1}) == Vec({'a','b','c'}, {'a':-1})
    True
    """
        return Vec(self.D, {key:-self[key] for key in self.D})

    def __rmul__(self, alpha):
        """
    Returns the scalar-vector product alpha times v.

    >>> zero = Vec({'x','y','z','w'}, {})
    >>> u = Vec({'x','y','z','w'},{'x':1,'y':2,'z':3,'w':4})
    >>> 0*u == zero
    True
    >>> 1*u == u
    True
    >>> 0.5*u == Vec({'x','y','z','w'},{'x':0.5,'y':1,'z':1.5,'w':2})
    True
    >>> u == Vec({'x','y','z','w'},{'x':1,'y':2,'z':3,'w':4})
    True
    """
        return Vec(self.D, {key : alpha*self[key] for key in self.D })


    def __mul__(self,other):
        #If other is a vector, returns the dot product of self and other
        if isinstance(other, Vec):
            return dot(self,other)
        else:
```

```
                return NotImplemented  #  Will cause other.__rmul__(self) to be invoked

        def __truediv__(self,other):   # Scalar division
            return (1/other)*self

        def __add__(self, other):
            """
        Returns the sum of the two vectors.

        Make sure to add together values for all keys from u.f and v.f even if some keys in \
u.f do not
        exist in v.f (or vice versa)

        >>> a = Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2})
        >>> b = Vec({'a','e','i','o','u'}, {'o':4,'u':7})
        >>> c = Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2,'o':4,'u':7})
        >>> a + b == c
        True
        >>> a == Vec({'a','e','i','o','u'}, {'a':0,'e':1,'i':2})
        True
        >>> b == Vec({'a','e','i','o','u'}, {'o':4,'u':7})
        True
        >>> d = Vec({'x','y','z'}, {'x':2,'y':1})
        >>> e = Vec({'x','y','z'}, {'z':4,'y':-1})
        >>> f = Vec({'x','y','z'}, {'x':2,'y':0,'z':4})
        >>> d + e == f
        True
        >>> d == Vec({'x','y','z'}, {'x':2,'y':1})
        True
        >>> e == Vec({'x','y','z'}, {'z':4,'y':-1})
        True
        >>> b + Vec({'a','e','i','o','u'}, {}) == b
        True
        """
            assert self.D == other.D
            return Vec(self.D, {key: self[key] + other[key] for key in self.D})

        def __radd__(self, other):
            "Hack to allow sum(...) to work with vectors"
            if other == 0:
                return self

        def __sub__(a,b):
            "Returns a vector which is the difference of a and b."
            return a+(-b)


        def __eq__(self, other):
            """
        Return true iff u is equal to v.

        Consider using brackets notation u[...] and v[...] in your procedure
        to access entries of the input vectors.  This avoids some sparsity bugs.

        >>> Vec({'a', 'b', 'c'}, {'a':0}) == Vec({'a', 'b', 'c'}, {'b':0})
        True
        >>> Vec({'a', 'b', 'c'}, {'a': 0}) == Vec({'a', 'b', 'c'}, {})
        True
        >>> Vec({'a', 'b', 'c'}, {}) == Vec({'a', 'b', 'c'}, {'a': 0})
```

```
        True

        Be sure that equal(u, v) checks equalities for all keys from u.f and v.f even if
        some keys in u.f do not exist in v.f (or vice versa)

        >>> Vec({'x','y','z'},{'y':1,'x':2}) == Vec({'x','y','z'},{'y':1,'z':0})
        False
        >>> Vec({'a','b','c'}, {'a':0,'c':1}) == Vec({'a','b','c'}, {'a':0,'c':1,'b':4})
        False
        >>> Vec({'a','b','c'}, {'a':0,'c':1,'b':4}) == Vec({'a','b','c'}, {'a':0,'c':1})
        False

        The keys matter:
        >>> Vec({'a','b'},{'a':1}) == Vec({'a','b'},{'b':1})
        False

        The values matter:
        >>> Vec({'a','b'},{'a':1}) == Vec({'a','b'},{'a':2})
        False
        """
            assert self.D == other.D
            return all([self[key] == other[key] for key in self.D])

        def is_almost_zero(self):
            s = 0
            for x in self.f.values():
                if isinstance(x, int) or isinstance(x, float):
                    s += x*x
                elif isinstance(x, complex):
                    y = abs(x)
                    s += y*y
                else: return False
            return s < 1e-20

        def __str__(v):
            "pretty-printing"
            D_list = sorted(v.D, key=repr)
            numdec = 3
            wd = dict([(k,(1+max(len(str(k)), len('{0:.{1}G}'.format(v[k], numdec))))) if is
ance(v[k], int) or isinstance(v[k], float) else (k,(1+max(len(str(k)), len(str(v[k]))))) for k in D
list])
            s1 = ''.join(['{0:>{1}}'.format(str(k),wd[k]) for k in D_list])
            s2 = ''.join(['{0:>{1}.{2}G}'.format(v[k],wd[k],numdec) if isinstance(v[k], int)
isinstance(v[k], float) else '{0:>{1}}'.format(v[k], wd[k]) for k in D_list])
            return "\n" + s1 + "\n" + '-'*sum(wd.values()) +"\n" + s2

        def __hash__(self):
            "Here we pretend Vecs are immutable so we can form sets of them"
            h = hash(frozenset(self.D))
            for k,v in sorted(self.f.items(), key = lambda x:repr(x[0])):
                if v != 0:
                    h = hash((h, hash(v)))
            return h

        def __repr__(self):
            return "Vec(" + str(self.D) + "," + str(self.f) + ")"

        def copy(self):
            "Don't make a new copy of the domain D"
```

```
            return Vec(self.D, self.f.copy())

        def __iter__(self):
            raise TypeError('%r object is not iterable' % self.__class__.__name__)

    if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

5.4 Inheritance

Inheritance is one of the basic tenets of object oriented programming and python supports multiple inheritance just like C++. Inheritance provides a mechanism for creating new classes that specialise or modify a base class thereby introducing new functionality. We call the base class the parent class or the super class. An example of a class inheriting from a base class in python is given in the following example.

```
class Account:
    """base class for representing user accounts"""
    num_accounts = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1

    def del_account(self):
        Account.num_accounts -= 1

    def __getattr__(self, name):
        """handle attribute reference for non-existent attribute"""
        return "Hey I dont see any attribute called {}".format(name)

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return "Name={}, balance={}".format(self.name, self.balance)

class SavingsAccount(Account):

    def __init__(self, name, balance, rate):
        super().__init__(name, balance)
        self.rate = rate


    def __repr__(self):
            return "SavingsAccount({}, {}, {})".format(self.name, self.balance, self.rate)


>>>acct = SavingsAccount("Obi", 10, 1)
>>>repr(acct)
SavingsAccount(Obi, 10, 1)
```

The `super` keyword

The `super` keyword plays an integral part in python inheritance. In a single inheritance hierarchy, the `super` keyword is used to refer to the parent/super class without explicitly naming it. This is similar to the `super` method in Java. This comes into play when overriding a method and there is a need to also call the parent version of such method as shown in the above example in which the `__init__` method in the `SavingsAccount` class is overridden but the `__init__` method of the parent class is also called using the `super` method. The `super` keyword plays a more integral role in python inheritance when a multiple inheritance hierarchy exists.

Multiple Inheritance

In multiple inheritance, a class can have multiple parent classes. This type of hierarchy is strongly discouraged. One of the issues with this kind of inheritance is the complexity involved in properly resolving methods when called. Imagine a class, `D`, that inherits from two classes, `B` and `C` and there is a need to call a method from the parent classes however both parent classes implement the same method. *How is the order in which classes are searched for the method determined ?* A *Method Resolution Order* algorithm determines how a method is found in a class or any of the class' base classes. In Python, the resolution order is calculated at class definition time and stored in the class `__dict__` as the `__mro__` attribute. To illustrate this, imagine a class hierarchy with multiple inheritance such as that showed in the following example.

```
    >>> class A:
    ...       def meth(self): return "A"
    ...
    >>> class B(A):
    ...       def meth(self): return "B"
    ...
    >>> class C(A):
    ...       def meth(self): return "C"
    ...
    >>> class D(B, C):
    ...       def meth(self): return "X"
    ...
    >>>
    >>> D.__mro__
    (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
    >>>
```

To obtain an `mro`, the interpreter method resolution algorithm carries out a left to right depth first listing of all classes in the hierarchy. In the trivial example above, this results in the following class list, `[D, B, A, C, A, object]`. Note that all objects will inherit from the root `object` class if no parent class is supplied during class definition. Finally, for each class that occurs multiple times, all occurrences are removed except the last occurrence resulting in an *mro* of `[D, B, C, A, object]` for the previous class hierarchy. This result is the order in which classes would be searched for attributes for a given instance of `D`.

Cooperative method calls with `super`

This section will show the power of the `super` keyword in a multiple inheritance hierarchy. The class hierarchy from the previous section is used. This example is from the excellent write up by Guido Van Rossum on *Type Unification*. Imagine that `A` defines a method that is overridden by `B`, `C` and `D`. Suppose that there is a requirement that all the methods are called; the method may be a `save` method that saves an attribute for each type it is defined for, so missing any call will result in some unsaved data in the hierarchy. A combination of `super` and `__mro__` provide the ammunition for solving this problem. This solution is referred to as the *call-next* method by Guido van Rossum and is shown in the following snippet:

```
class A(object):
     def meth(self):
          "save A's data"
          print("saving A's data")

class B(A):
     def meth(self):
          "save B's data"
          super(B, self).meth()
          print("saving B's data")

class C(A):
     def meth(self):
          "save C's data"
          super(C, self).meth()
          print("saving C's data")

class D(B, C):
     def meth(self):
          "save D's data"
          super(D, self).meth()
          print("saving D's data")
```

When *self.meth()* is called by an instance of `D` for example, `super(D, self).meth()` will find and call `B.meth(self)`, since B is the first base class following `D` that defines `meth` in `D.__mro__`. Now in `B.meth`, `super(B, self).m()` is called and since `self` is an instance of `D`, the next class after `B` is `C` (`__mro__` is `[D, B, C, A]`) and the search for a definition of `meth` continues here. This finds `C.meth` which is called, and which in turn calls `super(C, self).m()`. Using the same *MRO*, the next class after `C` is `A`, and thus `A.meth` is called. This is the original definition of m, so no further *super()* call is made at this point. Using `super` and method resolution order, the interpreter has been able to find and call all version of the `meth` method implemented by each of the classes in the hierarchy. However, multiple inheritance is best avoided because for more complex class hierarchies, the calls may be way more complicated than this.

5.5 Static and Class Methods

All methods defined in a class by default operate on instances. However, one can define static or class methods by decorating such methods with the corresponding `@staticmethod` or `@classmethod` decorators.

Static Methods

Static methods are normal functions that exist in the name-space of a class. Referencing a static method from a class shows that rather than an *unbound* method type, a *function* type is returned as shown below:

```
class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt

     def inquiry(self):
          return "Name={}, balance={}".format(self.name, self.balance)

     @staticmethod
     def static_test_method():
          return "Current Account"

>>> Account.static_test_method
<function Account.static_test_method at 0x101b846a8>
```

To define a static method, the `@staticmethod` decorator is used and such methods do not require the `self` argument. Static methods provide a mechanism for better organization as code related to a class are placed in that class and can be overridden in a sub-class as needed. Unlike ordinary class methods that are wrappers around the actual underlying functions, static methods return the underlying functions without any modification when used.

Class Methods

Class methods as the name implies operate on classes themselves rather than instances. Class methods are created using the `@classmethod` decorator with the `class` rather than `instance` passed as the first argument to the method.

```
import json

class Account(object):
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1
```

```
    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt

    def inquiry(self):
        return "Name={}, balance={}".format(self.name, self.balance)

    @classmethod
    def from_json(cls, params_json):
                params = json.loads(params_json)
        return cls(params.get("name"), params.get("balance"))

    @staticmethod
    def type():
        return "Current Account"
```

A motivating example of the usage of class methods is as a *factory* for object creation. Imagine data for the `Account` class comes in different formats such as *tuples*, *json string* etc. It is not possible to define multiple `__init__` methods in a class so class methods come in handy for such situations. In the `Account` class defined above for example, there is a requirement to initialize an account from a *json* string object so we define a class factory method, `from_json` that takes in a json string object and handles the extraction of parameters and creation of the account object using the extracted parameters. Another example of a class method in action as a factory method is the `dict.fromkeys` methods that is used for creating *dict* objects from a sequence of supplied keys and value.

5.6 Descriptors and Properties

Descriptors are an esoteric but integral part of the python programming language. They are used widely in the core of the python language and a good grasp of descriptors provides a python programmer with a deeper understanding of the language. To set the stage for the discussion of descriptors, some scenarios that a programmer may encounter are described; this is followed by an explanation of descriptors and how they provide elegant solutions to these scenarios.

1. Consider a program in which some rudimentary type checking of object data attributes needs to be enforced. Python is a dynamic languages so does not support type checking but this does not prevent anyone from implementing a version of type checking regardless of how rudimentary it may be. The conventional way to go about type checking object attributes may take the following form.

```
    def __init__(self, name, age):
        if isinstance(str, name):
            self.name = name
        else:
            raise TypeError("Must be a string")
        if isinstance(int, age):
            self.age = age
        else:
            raise TypeError("Must be an int")
```

The above method maybe feasible for enforcing such type checking for one or two data attributes but as the attributes increase in number it gets cumbersome. Alternatively, a `type_check(type, val)` function could be defined and this will be called in the `__init__` method before assignment; but this cannot be elegantly applied when the attribute value is set after initialization. A quick solution that comes to mind is the `getters` and `setters` present in Java but that is un-pythonic and cumbersome.

2. Consider a program that needs object attributes to be read-only once initialized. One could also think of ways of implementing this using Python special methods but once again such implementation could be unwieldy and cumbersome.
3. Finally, consider a program in which the attribute access needs to be customized. This maybe to log such attribute access or to even perform some kind of transformation of the attribute for example. Once again, it is not too difficult to come up with a solution to this although such solution maybe unwieldy and not reusable.

All the above mentioned issues are all linked together by the fact that they are all related to attribute references. Attribute access is trying to be customized.

Enter Python Descriptors

Descriptors provide elegant, simple, robust and re-usable solutions to the above listed issues. Simply put, a *descriptor* is an **object** that represents the value of an attribute. This means that if an account object has an attribute `name`, a descriptor is another object that can be used to represent the value held by that attribute, `name`. Such an object implements the `__get__`, `__set__` or `__delete__` special methods of the descriptor protocol. The signature for each of these methods is shown below:

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Objects implementing only the `__get__` method are non-data descriptors so they can only be read from after initialization while objects implementing the `__get__` and `__set__` are data descriptors meaning that such descriptor objects are writable.

To get a better understanding of descriptors descriptor based solutions are provided to the issues mentioned in the previous section. Implementing type checking on an object attribute using descriptors is a simple and straightforward task. A decorator implementing this type checking is shown in the following snippet.

```
class TypedAttribute:

    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self,instance,value):
```

```
            if not isinstance(value,self.type):
                raise TypeError("Must be a %s" % self.type)
            setattr(instance,self.name,value)

        def __delete__(self,instance):
            raise AttributeError("Can't delete attribute")


    class Account:
        name = TypedAttribute("name",str)
        balance = TypedAttribute("balance",int, 42)

    >> acct = Account()
    >> acct.name = "obi"
    >> acct.balance = 1234
    >> acct.balance
    1234
    >> acct.name
    obi
    # trying to assign a string to number fails
    >> acct.balance = '1234'
    TypeError: Must be a <type 'int'>
```

In the example, a descriptor, `TypedAttribute` is implemented and this descriptor class enforces rudimentary type checking for any attribute of a class which it is used to represent. It is important to note that descriptors are effective in this kind of case only when defined at the class level rather than instance level i.e. in `__init__` method as shown in the example above.

Descriptors are integral to the Python language. Descriptors provide the mechanism behind properties, static methods, class methods, `super` and a host of other functionality in Python classes. In fact, descriptors are the first type of object searched for during an attribute reference. When an object is referenced, a reference, `b.x`, is transformed into `type(b).__dict__['x'].__get__(b, type(b))`. The algorithm then searches for the attribute in the following order.

1. `type(b).__dict__` is searched for the attribute name and if a data descriptor is found, the result of calling the descriptor's `__get__` method is returned. If it is not found, then all base classes of `type(b)` are searched in the same way.
2. `b.__dict__` is searched and if attribute name is found here, it is returned.
3. `type(b).__dict` is searched for a non-data descriptor with given attribute name and if found it is returned,
4. If the name is not found, an `AttributeError` is raised or `__getattr__()` is called if provided.

This precedence chain can be overridden by defining custom `__getattribute__` methods for a given object class (the precedence defined above is contained in the default `__getattribute__` provided by the interpreter).

With a firm understanding of the mechanics of descriptors, it is easy to implement elegant solutions to the second and third issues raised in the previous section. Implementing a read only attribute with descriptors becomes a simple case of implementing a non-data

descriptor *i.e descriptor with no `__set__` method*. To solve the custom access issue, whatever functionality is required is added to the `__get__`and `__set__` methods respectively.

Class Properties

Defining descriptor classes each time a descriptor is required is cumbersome. Python ***properties*** provide a concise way of adding data descriptors to attributes. A property signature is given below:

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

`fget`, `fset` and `fdel` are the *getter, setter and deleter* methods for such class attributes. The process of creating properties is illustrated with the following example.

```
class Accout(object):
     def __init__(self):
          self._acct_num = None

     def get_acct_num(self):
          return self._acct_num

     def set_acct_num(self, value):
          self._acct_num = value

     def del_acct_num(self):
          del self._acct_num

     acct_num = property(get_acct_num, set_acct_num, del_acct_num, "Account number property.")
```

If `acct` is an instance of `Account`, `acct.acct_num` will invoke the getter, `acct.acct_num = value` will invoke the setter and `del acct_num.acct_num` will invoke the deleter.

The property object and functionality can be implemented in python as illustrated in Descriptor How-To Guide using the descriptor protocol as shown below :

```
class Property(object):
     "Emulate PyProperty_Type() in Objects/descrobject.c"

     def __init__(self, fget=None, fset=None, fdel=None, doc=None):
          self.fget = fget
          self.fset = fset
          self.fdel = fdel
          if doc is None and fget is not None:
               doc = fget.__doc__
          self.__doc__ = doc

     def __get__(self, obj, objtype=None):
          if obj is None:
               return self
          if self.fget is None:
               raise AttributeError("unreadable attribute")
          return self.fget(obj)

     def __set__(self, obj, value):
          if self.fset is None:
```

```
                raise AttributeError("can't set attribute")
            self.fset(obj, value)

        def __delete__(self, obj):
            if self.fdel is None:
                raise AttributeError("can't delete attribute")
            self.fdel(obj)

        def getter(self, fget):
            return type(self)(fget, self.fset, self.fdel, self.__doc__)

        def setter(self, fset):
            return type(self)(self.fget, fset, self.fdel, self.__doc__)

        def deleter(self, fdel):
            return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

Python also provides the `@property` decorator that can be used to create read only attributes. A property object has getter, setter, and deleter decorator methods that can be used to create a copy of the property with the corresponding *accessor* function set to the decorated function. This is best explained with an example:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
     # the x property. the decorator creates a read-only property
    def x(self):
        return self._x

    @x.setter
    # the x property setter makes the property writeable
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

If a property is read-only then the `setter` method is left out.

An understanding of descriptors puts us in a better corner to understand what actually goes on during a method call. Note that methods are stored as ordinary functions in a class dictionary as shown in the following snippet.

```
>>>Account.inquiry
<function Account.inquiry at 0x101a3e598>
>>>
```

However, object methods are of bound method type as shown in the following snippet.

```
>>> x = Account("nkem", 10)
>>> x.inquiry
<bound method Account.inquiry of <Account object at 0x101a3c588>>
```

To understand how this transformation takes place, note that a bound method is just a thin wrapper around the class function. Functions are descriptors because they have the `__get__` method attribute so a reference to a function will result in a call to the `__get__` method of the function and this returns the desired type, the function itself or a bound method, depending on whether this reference is from a class or from an instance of the class. It is not difficult to imagine how static and class methods maybe implemented by the function descriptor and this is left to the reader to come up with.

5.7 Abstract Base Classes

Sometimes, it is necessary to enforce a contract between classes in a program. For example, it may be necessary for all classes to implement a set of methods. This is accomplished using interfaces and abstract classes in statically typed languages like Java. In Python, a base class with default methods may be implemented and then all other classes within the set inherit from the base class. However, there is a requirement for each each subclass to have its own implementation and this rule needs to be enforced. All the needed methods can be defined in a base class with each of them having an implementation that raises the `NotImplementedError` exception. All subclasses then have to override these methods in order to use them. However this does not still solve the problem fully. It is possible that some subclasses may not implement some of these method and it would not be known till a method call was attempted at runtime.

Consider another situation of a proxy object that passes method calls to another object. Such a proxy object may implement all required methods of a type via its proxied object, but an `isinstance` test on such a proxy object for the proxied object will fail to produce the correct result.

Python's *Abstract base classes* provide a simple and elegant solution to these issues mentioned above. The abstract base class functionality is provided by the *abc* module. This module defines a meta-class (we discuss meta-classes in the chapter on meta-programming) and a set of decorators that are used in the creation of abstract base classes. When defining an abstract base class we use the `ABCMeta` meta-class from the `abc` module as the meta-class for the abstract base class and then make use of the `@abstractmethod` and `@abstractproperty` decorators to create methods and properties that must be implemented by non-abstract subclasses. If a subclass doesn't implement any of the abstract methods or properties then it is also an abstract class and cannot be instantiated as illustrated below:

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
     __meta-class__ = ABCMeta

     @abstractmethod
     def change_gear(self):
          pass

     @abstractmethod
     def start_engine(self):
          pass
```

```
class Car(Vehicle):

     def __init__(self, make, model, color):
          self.make = make
          self.model = model
          self.color = color

# abstract methods not implemented
>>> car = Car("Toyota", "Avensis", "silver")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Car with abstract methods change_gear, start_engine
>>>
```

Once, a class implements all abstract methods then that class becomes a concrete class and can be instantiated by a user.

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
     __meta-class__ = ABCMeta

     @abstractmethod
     def change_gear(self):
          pass

     @abstractmethod
     def start_engine(self):
          pass


class Car(Vehicle):

     def __init__(self, make, model, color):
          self.make = make
          self.model = model
          self.color = color

     def change_gear(self):
          print("Changing gear")

     def start_engine(self):
          print("Changing engine")

>>> car = Car("Toyota", "Avensis", "silver")
>>> print(isinstance(car, Vehicle))
True
```

Abstract base classes also allow existing classes to register as part of its hierarchy but it performs no check on whether such classes implement all the methods and properties that have been marked as abstract. This provides a simple solution to the second issue raised in the opening paragraph. Now, a proxy class can be registered with an abstract base class and `isinstance` check will return the correct answer when used.

```
from abc import ABCMeta, abstractmethod
```

```
class Vehicle(object):
     __meta-class__ = ABCMeta

     @abstractmethod
     def change_gear(self):
          pass

     @abstractmethod
     def start_engine(self):
          pass


class Car(object):

     def __init__(self, make, model, color):
          self.make = make
          self.model = model
          self.color = color

>>> Vehicle.register(Car)
>>> car = Car("Toyota", "Avensis", "silver")
>>> print(isinstance(car, Vehicle))
True
```

Abstract base classes are used a lot in python library. They provide a mean to group python objects such as number types that have a relatively flat hierarchy. The `collections` module also contains abstract base classes for various kinds of operations involving sets, sequences and dictionaries. Whenever we want to enforce contracts between classes in python just as interfaces do in Java, abstract base classes is the way to go.

6. The Function

The function is another organizational unit of code in Python. Python functions are either named or *anonymous* set of statements or expressions. In Python, ***functions are first class objects***. This means that there is no restriction on function use as values; introspection on functions can be carried out, functions can be assigned to variables, functions can be used as arguments to other function and functions can be returned from method or function calls just like any other python value such as strings and numbers.

6.1 Function Definitions

The `def` keyword is the usual way of creating user-defined functions. Function definitions are executable statements.

```
def square(x):
    return x**2
```

When a function definition such as the `square` function defined above is encountered, only the function definition statement, *that is* `def square(x)`, is executed; this implies that all arguments are evaluated. The evaluation of arguments has some implications for function default arguments that have mutable data structure as values; this will be covered later on in this chapter. The execution of a function definition binds the function name in the current name-space to a function object which is a wrapper around the executable code for the function. This function object contains a reference to the current global name-space which is the global name-space that is used when the function is called. The function definition does not execute the function body; this gets executed only when the function is called.

Python also has support for ***anonymous functions***. These functions are created using the `lambda` keyword. Lambda expressions in python are of the form:

```
lambda_expr ::=  "lambda" [parameter_list]: expression
```

Lambda expressions return function objects after evaluation and have same attributes as named functions. Lambda expressions are normally only used for very simple functions in python due to the fact that a lambda definition can contain only one expression. A lambda definition for the square function defined above is given in the following snippet.

```
>>> square = lambda x: x**2
>>> for i in range(10):
    square(i)
0
1
4
9
16
25
```

```
36
49
64
81
>>>
```

6.2 Functions are Objects

Functions just like other values are objects. Functions have the type `function`.

```
>>> def square(x):
...       return x*x

>>> type(square)
<class 'function'>
```

Like every other object, introspection on functions using the `dir()` function provides a list of function attributes.

```
def square(x):
  return x**2

>>> square
<function square at 0x031AA230>
>>> dir(square)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__dela
tr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribu
e__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__mod
le__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
```

Some important function attributes include:

- `__annotations__` this attribute contains optional meta-data information about arguments and return types of a function definition. Python 3 introduced the optional annotation functionality primarily to help tools used in developing python software. An example of a function annotation is shown in the following example.

  ```
  >>> def square(x: int) -> int:
  ...       return x*x
  ...
  square.__annotations__
  {'x': <class 'int'>, 'return': <class 'int'>}
  ```

  Parameters are annotated by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return values are annotated by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. In the case of default values for functions, the annotation is of the following form.

  ```
  >>> def def_annotation(x: int, y: str = "obi"):
  ...       pass
  ```

- `__doc__` returns the documentation string for the given function.

```
def square(x):
  """return square of given number"""
  return x**2

>>> square.__doc__
'return square of given number'
```

- `__name__` returns function name.

```
>>> square.func_name
'square'
```

- `__module__` returns the name of module function is defined in.

```
>>> square.__module__
'__main__'
```

- `__defaults__` returns a tuple of the default argument values. Default arguments are discussed later on.
- `__kwdefaults__` returns a `dict` containing default keyword argument values.
- `__globals__` returns a reference to the dictionary that holds the function's global variables (see the chapter 5 for a word on global variables).

```
>>> square.func_globals
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'square': <
n square at 0x10f099c08>, '__doc__': None, '__package__': None}
```

- `__dict__` returns the name-space supporting arbitrary function attributes.

```
>>> square.func_dict
{}
```

- `__closure__` returns tuple of cells that contain bindings for the function's free variables. Closures are discussed later on in this chapter.

Functions can be passed as arguments to other functions. These functions that take other functions as argument are commonly referred to as ***higher order*** functions and these form a very important part of ***functional programming***. A very good example of a higher order function is the `map` function that takes a *function* and an `iterable` and applies the *function* to each item in the `iterable` returning a new list. In the following example, we illustrate the use of the `map()` higher order function by passing the `square` function previously defined and an `iterable` of numbers to the `map` function.

```
>>> map(square, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A function can be defined inside another function as well as returned from a function call.

```
>>> def make_counter():
...       count = 0
...       def counter():
```

```
...             nonlocal count # nonlocal captures the count binding from enclosing scope not
obal scope
...             count += 1
...             return count
...         return counter
```

In the previous example, a function, `counter` is defined within another function, `make_counter`, and the `counter` function is returned whenever the `make_counter` function is executed. Functions can also be assigned to variables just like any other python object as shown below:

```
>>> def make_counter():
...         count = 0
...         def counter():
...             nonlocal count # nonlocal captures the count binding from enclosing scope not glol
 scope
...             count += 1
...             return count
...         return counter
>>> func = make_counter()
>>> func
<function inner at 0x031AA270>
>>>
```

In the above example, the `make_counter` function returns a function when called and this is assigned to the variable `func`. This variable refers to a function object and can be called just like any other function as shown in the following example:

```
>>> func()
1
```

6.3 Functions are *descriptors*

As mentioned in the previous chapter, functions are also descriptors. An inspection of the attributes of a function as shown in the following example shows that a function has the `__get__` method attribute thus making them *non-data descriptors*.

```
>>> def square(x):
...         return x**2
...
>>> dir(square) # see the __get__ attribute
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__dela
tr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribu
e__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__mod
le__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
```

This `__get__` method is called whenever a function is referenced and provides the mechanism for handling method calls from objects and ordinary function calls. This descriptor characteristic of a function enables functions to return either itself or a *bound method*/ when referenced depending on where and how it is referenced.

6.4 Calling Functions

In addition to calling functions in the conventional way with normal arguments, Python also supports functions with variable number of arguments. These variable number of arguments come in three flavours that are described below:

- **Default Argument Values**: This allows a user to define default values for some or all function arguments. In this case, such a function can be called with fewer arguments and the interpreter will use default values for arguments that are not supplied during function call. This following example is illustrative.

```
def show_args(arg, def_arg=1, def_arg2=2):
     return "arg={}, def_arg={}, def_arg2={}".format(arg, def_arg, def_arg2)
```

  The above function has been defined with a single normal positional argument, `arg` and two default arguments, `def_arg` and `def_arg2`. The function can be called in any of the following ways below:

  - Supplying non-default positional argument values only; in this case the other arguments take on the supplied default values:

```
def show_args(arg, def_arg=1, def_arg2=2):
     return "arg={}, def_arg={}, def_arg2={}".format(arg, def_arg, def_arg2)

>>> show_args("tranquility")
'arg=tranquility, def_arg=1, def_arg2=2'
```

  - Supplying values to override some default arguments in addition to the non-default positional arguments:

```
def show_args(arg, def_arg=1, def_arg2=2):
     return "arg={}, def_arg={}, def_arg2={}".format(arg, def_arg, def_arg2)

>>> show_args("tranquility", "to Houston")
'arg=tranquility, def_arg=to Houston, def_arg2=2'
```

  - Supplying values for all arguments overriding even arguments with default values.

```
def show_args(arg, def_arg=1, def_arg2=2):
     return "arg={}, def_arg={}, def_arg2={}".format(arg, def_arg, def_arg2)

>>> show_args("tranquility", "to Houston", "the eagle has landed")
'arg=tranquility, def_arg=to Houston, def_arg2=the eagle has landed'
```

  It is also very important to be careful when using mutable data structures as default arguments. Function definitions get executed only once so these mutable data structures are created once at definition time. This means that the same mutable data structure is used for all function calls as shown in the following example:

```
def show_args_using_mutable_defaults(arg, def_arg=[]):
     def_arg.append("Hello World")
     return "arg={}, def_arg={}".format(arg, def_arg)

>>> show_args_using_mutable_defaults("test")
"arg=test, def_arg=['Hello World']"
```

```
>>> show_args_using_mutable_defaults("test 2")
"arg=test 2, def_arg=['Hello World', 'Hello World']"
```

On every function call, `Hello World` is added to the `def_arg` list and after two function calls the default argument has two hello world strings. It is important to take note of this when using mutable default arguments as default values.

- **Keyword Argument**: Functions can be called using keyword arguments of the form `kwarg=value`. A `kwarg` refers to the name of arguments used in a function definition. Take the function defined below with positional non-default and default arguments.

```
def show_args(arg, def_arg=1):
     return "arg={}, def_arg={}".format(arg, def_arg)
```

To illustrate function calls with key word arguments, the following function can be called in any of the following ways:

```
show_args(arg="test", def_arg=3)

show_args(test)

show_args(arg="test")

show_args("test", 3)
```

In a function call, keyword arguments must not come before non-keyword arguments thus the following will fail:

```
show_args(def_arg=4)
```

A function cannot supply duplicate values for an argument so the following is illegal:

```
show_args("test", arg="testing")
```

In the above the argument `arg` is a positional argument so the value `test` is assigned to it. Trying to assign to the keyword `arg` again is an attempt at multiple assignment and this is illegal.

All the keyword arguments passed must match one of the arguments accepted by the function and the order of keywords including non-optional arguments is not important so the following in which the order of argument is switched is legal:

```
show_args(def_arg="testing", arg="test")
```

- **Arbitrary Argument List**: Python also supports defining functions that take arbitrary number of arguments that are passed to the function in a tuple. An example of this from the python tutorial is given below:

```
def write_multiple_items(file, separator, *args):
     file.write(separator.join(args))
```

The arbitrary number of arguments must come after normal arguments; in this case, after the `file` and `separator` arguments. The following is an example of function calls to the above defined function:

```
f = open("test.txt", "wb")
write_multiple_items(f, " ", "one", "two", "three", "four", "five")
```

The arguments `one two three four five` are all bunched together into a tuple that can be accessed via the `args` argument.

Unpacking Function Argument

Sometimes, arguments for a function call are either in a tuple, a list or a `dict`. These arguments can be unpacked into functions for function calls using `*` or `**` operators. Consider the following function that takes two positional arguments and prints out the values

```
def print_args(a, b):
     print a
     print b
```

If the values for a function call are in a list then these values can be unpacked directly into the function as shown below:

```
>>> args = [1, 2]
>>> print_args(*args)
1
2
```

Similarly, dictionaries can be used to store `keyword to value` mapping and the `**` operator is used to unpack the keyword arguments to the functions as shown below:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
              print "-- This parrot wouldn't", action,
              print "if you put", voltage, "volts through it.",
              print "E's", state, "!"

>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
>>> This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised
```

`*` and `**` Function Parameters

Sometimes, when defining a function, it is not known before hand the number of arguments to expect. This leads to function definitions of the following signature:

```
show_args(arg, *args, **kwargs)
```

The `*args` argument represents an unknown length of sequence of positional arguments while `**kwargs` represents a dict of keyword name value mappings which may contain any amount of keyword name value mapping. The `*args` must come before `**kwargs` in the function definition. The following illustrates this:

```
def show_args(arg, *args, **kwargs):
     print arg
     for item in args:
          print args
     for key, value in kwargs:
          print key, value

>>> args = [1, 2, 3, 4]
>>> kwargs = dict(name='testing', age=24, year=2014)
>>> show_args("hey", *args, **kwargs)
hey
1
2
3
4
age 24
name testing
year 2014
```

The normal argument must be supplied to the function but the `*args` and `**kwargs` are optional as shown below:

```
>>> show_args("hey", *args, **kwargs)
hey
```

At function call the normal argument(s) is/are supplied normally while the optional arguments are unpacked. This kind of function definition comes in handy when dealing with function decorators as will be seen in the chapter on decorators.

6.5 Nested functions and Closures

Function definitions within another function creates nested functions as shown in the following snippet.

```
>>> def make_counter():
...       count = 0
...       def counter():
...            nonlocal count # nonlocal captures the count binding from enclosing scope not glol
 scope
...            count += 1
...            return count
...       return counter
...
```

In the nested function definition, the function `counter` is in scope only inside the function `make_counter`, so it is often useful when the `counter` function is returned from the `make_counter` function. In nested functions such as in the above example, a new instance of the nested function is created on each call to outer function. This is because during each execution of the `make_counter` function, the definition of the `counter` function is executed but the body is not executed.

A nested function has access to the environment in which it was created. A result is that a variable defined in the outer function can be referenced in the inner function even after the outer functions has finished execution.

```
>>> x = make_counter()
>>> x
<function counter at 0x0273BCF0>
>>> x()
1
```

When nested functions reference variables from the outer function in which they are defined, the nested function is said to be closed over the referenced variable. The `__closure__` special attribute of a function object is used to access the closed variables as shown in the next example.

```
>>> cl = x.__closure__
>>> cl
(<cell at 0x029E4470: str object at 0x02A0FD90>,)

>>> cl[0].cell_contents
0
```

Closures in previous versions of Python have a quirky behaviour. In Python *2.x* and below, variables that reference immutable types such as string and numbers cannot be rebound within a closure. The following example illustrates this.

```
def counter():
     count = 0
     def c():
          count += 1
          return count
     return c

>>> c = counter()
>>> c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in c
UnboundLocalError: local variable 'count' referenced before assignment
```

A rather wonky solution to this is to make use of a mutable type to capture the closure as shown below:

```
def counter():
     count = [0]
     def c():
          count[0] += 1
          return count[0]
     return c

>>> c = counter()
>>> c()
1
>>> c()
2
>>> c()
3
```

Python 3 introduced the `nonlocal` key word that fixed this closure scoping issue as shown in the following snippet.

```
def counter():
     count = 0
     def c():
          nonlocal count
          count += 1
          return count
      return c
```

Closures can be used for maintaining states ***(isn't that what classes are for)*** and for some simple cases provide a more succinct and readable solution than classes. A class version of a logging API tech_pro is shown in the following example.

```
class Log:
     def __init__(self, level):
          self._level = level

     def __call__(self, message):
          print("{}: {}".format(self._level, message))

log_info = Log("info")
log_warning = Log("warning")
log_error = Log("error")
```

The same functionality that the class based version possesses can be implemented with functions closures as shown in the following snippet:

```
def make_log(level):
     def _(message):
          print("{}: {}".format(level, message))
     return _

log_info = make_log("info")
log_warning = make_log("warning")
log_error = make_log("error")
```

The closure based version as can be seen is way more succinct and readable even though both versions implement exactly the same functionality. Closures also play a major role in a major ***function decorators***. This is a widely used functionality that is explained in the chapter on meta-programming. Closures also form the basis for the `partial` function, a function that is described in detail in the next section. With a firm understanding of functions, a tour of some techniques and modules for functional programming in Python is given in the following section.

6.6 A Byte of Functional Programming

The Basics

The hallmark of functional programming is the absence of side effects in written code. This essentially means that in the functional style of programming object values do not change once they are created and to reflect a *change* in an object value, a new object with

the changed value is created. An example of a function with side effects is the following snippet in which the original argument is modified and then returned.

```
def squares(numbers):
    for i, v in enumerate(numbers):
        numbers[i] = v**2
    return numbers
```

A functional version of the above would avoid any modification to arguments and create new values that are then returned as shown in the following example.

```
def squares(numbers):
    return map(lambda x:x*x, numbers)
```

Language features such as first class functions make functional programming possible while programming techniques such as `mapping`, `reducing`, `filtering`, `currying` and `recursion` are examples of techniques for implementing a functional style of programming. In the above example, the `map` function applies the function `lambda x:x*x` to each element in the supplied sequence of numbers.

Python provides built-in functions such as `map`, `filter` and `reduce` that aid in functional programming. A description of these functions follows.

1. `map(func, iterable)`: This is a classic functional programming construct that takes a function and an iterable as argument and returns an iterator that applies the function to each item in the iterable. The `squares` function from above is an illustration of `map` in use. The ideas behind the `map` and `reduce` constructs have seen application in large scale data processing with the popular `MapReduce` programming model that is used to fan out (`map`) operation on large data streams to a cluster of distributed machines for computation and then gather the result of these computations together (`reduce`).
2. `filter(func, iterable)`: This also takes a function and an iterable as argument. It returns an iterator that applies `func` to each element of the iterable and returns elements of the iterable for which the result of the application is `True`. The following trivial example selects all even numbers from a list.

```
>>> even = lambda x: x%2==0
>>> even(10)
True
>>> filter(even, range(10))
<filter object at 0x101c7b208>
>>> list(filter(even, range(10)))
[0, 2, 4, 6, 8]
```

3. `reduce(func, iterable[, initializer])`: This is no longer a built-in and was moved into the `functools` modules in Python 3 but is discussed here for completeness. The `reduce` function applies `func` cumulatively to the items in `iterable` in order to get a single value that is returned. `func` is a function that takes two positional arguments. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `(((1+2)+3)+4)+5)`; it starts out reducing the first two arguments then reduces the third with the result of the first two and so on. If the optional initializer is

provided, then it serves as the base case. An illustration of this is flattening a nested list which we illustrate below.

```
>>> import functools
>>> def flatten_list(nested_list):
...      return functools.reduce(lambda x, y: x + y, nested_list, [])
...
>>> flatten_list([[1, 3, 4], [5,6, 7], [8, 9, 10]])
[1, 3, 4, 5, 6, 7, 8, 9, 10]
```

The above listed functions are examples of built-in higher order functions in Python. Some of the functionality they provide can be replicated using more common constructs. Comprehensions are one of the most popular alternatives to these higher order functions.

Comprehensions

Python comprehensions are syntactic constructs that enable sequences to be built from other sequences in a clear and concise manner. Python comprehensions are of three types namely:

1. List Comprehensions.
2. Set Comprehensions.
3. Dictionary Comprehensions.

List Comprehensions

List comprehensions are by far the most popular Python comprehension construct. List comprehensions provide a concise way to create new list of elements that satisfy a given condition from an **iterable**. A list of squares for a sequence of numbers can be computed using the following `squares`function that makes use of the `map` function.

```
def squares(numbers):
     return map(lambda x:x*x, numbers)
>>> sq = squares(range(10))
```

The same list can be created in a more concise manner by using list comprehensions rather than the `map` function as in the following example.

```
>>> squares = [x**2 for x in range(10)]
```

The comprehension version is clearer and more concise than the conventional `map` method for one without any experience in higher order functions.

According to the python documentation,

> **a list comprehension consists of square brackets containing an expression followed by a *for* clause and zero or more for or *if* clauses.**

```
[expression for item1 in iterable1 if condition1
              for item2 in iterable2 if condition2
              ...
              for itemN in iterableN if conditionN ]
```

The result of a list comprehension expression is a new list that results from evaluating the expression in the context of the *for* and *if* clauses that follow it. For example, to create a list of the squares of even numbers between 0 and 10, the following comprehension is used.

```
>>> even_squares = [i**2 for i in range(10) if i % 2 == 0]
>>> even_squares
[0, 4, 16, 36, 64]
```

The expression `i**2` is computed in the context of the *for* clause that iterates over the numbers from 0 to 10 and the *if* clause that filters out non-even numbers.

Nested *for* loops and List Comprehensions

List comprehensions can also be used with multiple or nested *for* loops. Consider for example, the simple code fragment shown below that creates a tuple from pair of numbers drawn from the two sequences given.

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

The above can be rewritten in a more concise and simple manner as shown below using list comprehensions

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

It is important to take into consideration the order of the *for* loops as used in the list comprehension. Careful observation of the code snippets using comprehension and that without comprehension shows that the order of the *for* loops in the comprehension follows the same order if it had been written without comprehensions. The same applies to nested for loops with nesting depth greater than two.

Nested List Comprehensions

List comprehensions can also be nested. Consider the following example drawn from the Python documentation of a `3x4` matrix implemented as a list of 3 lists each of length 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Transposition is a matrix operation that creates a new matrix from an old one using the rows of the old matrix as the columns of the new matrix and the columns of the old matrix as the rows of the new matrix. The rows and columns of the matrix can be transposed using the following nested list comprehension:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

The above is equivalent to the following snippet.

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Set Comprehensions

In set comprehensions, braces rather than square brackets are used to create new sets. For example, to create the set of the squares of all numbers between 0 and 10, the following set comprehensions is used.

```
>>> x = {i**2 for i in range(10)}
>>> x
set([0, 1, 4, 81, 64, 9, 16, 49, 25, 36])
>>>
```

Dict Comprehensions

Braces are also used to create new dictionaries in `dict` comprehensions. In the following example, a mapping of a number to its square is created using `dict` comprehensions.

```
>>> x = {i:i**2 for i in range(10)}
>>> x
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Functools

The `functools` module in Python contains a few *higher order* functions that act on and return other functions. A few of the interesting *higher order* functions that are included in this module are described.

1. `partial(func, *args, **keywords)` This is a function that when called returns an object that can be called like the original `func` argument with `*args` and `**keywords` as arguments. If the returned object is called with additional `*args` or `**keyword` arguments then these are added to the original `*args` and `**keywords` and the updated set of arguments are used in the function call. This is illustrated with the following trivial example.

   ```
   >>> from functools import partial
   >>> basetwo = partial(int, base=2)
   >>> basetwo.__doc__ = 'Convert base 2 string to an int.'
   >>> basetwo('10010')
   18
   ```

   In the above example, a new *callable*, `basetwo`, that takes a number in binary and converts it a number in decimal is created. What has happened is that the `int()` functions that takes two arguments has been wrapped by a callable, `basetwo` that

takes only one argument. To understand how this may work, take your mind back to the discussion about closures and how variable captures work. Once this is understood, it is easy to imagine how to implement this partial function. The partial function has functionality that is equivalent to the following closure as defined in the Python documentation.

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

Partial objects provide elegant solutions to some practical problems that are encountered during development. For example, suppose one has a list of points represented as tuples of `(x,y)` coordinates and there is a requirement to sort all the points according to their distance from some other central point. The following function computes the distance between two points in the *xy* plane:

```
>>> points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]
>>> import math
>>> def distance(p1, p2):
...     x1, y1 = p1
...     x2, y2 = p2
...     return math.hypot(x2 - x1, y2 - y1)
```

The built-in `sort()` method of lists is handy here and accepts a key argument that can be used to customize sorting, but it only works with functions that take a single argument thus `distance()` is unsuitable. The `partial` method provides an elegant method of dealing with this as shown in the following snippet.

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

The `partial` function creates and returns a callable that takes a single argument, a point. Now note that the partial object has captured the reference point, `pt` already so when the key is called with the point argument, the distance function passed to the partial function is used to compute the distance between the supplied point and the reference point.

2. `@functools.lru_cache(maxsize=128, typed=False)`: This decorator is used to wrap a function with a memoizing callable that saves up to the `maxsize` number of most recent calls. When `maxsize` is reached, oldest cached values are ejected. Caching can save time when an expensive or I/O bound function is periodically called with the same arguments. This decorator makes use of a `dictionary` for

storing results so is limited to caching only arguments that are hashable. The `lru_cache` decorator provides a function, the `cache_info` for stats on cache useage.

3. `@functools.singledispatch`: This is a decorator that changes a function into a single dispatch generic function. The functionality aims to handle dynamic overloading in which a single function can handle multiple types. The mechanics of this is illustrated with the following code snippet.

```
@singledispatch
def fun(arg, verbose=False):
     if verbose:
          print("Let me just say,", end=" ")
     print(arg)

@fun.register(int)
def _(arg, verbose=False):
     if verbose:
          print("Strength in numbers, eh?", end=" ")
     print(arg)

@fun.register(list)
def _(arg, verbose=False):
     if verbose:
          print("Enumerate this:")
     for i, elem in enumerate(arg):
          print(i, elem)

fun("Hello, world.")
fun(1, verbose=True)
fun([1, 2, 3], verbose=True)
fun((1, 2, 3), verbose=True)

Hello, world.
Strength in numbers, eh? 1
Enumerate this:
0 1
1 2
2 3
Let me just say, (1, 2, 3)
```

A generic function is defined with the `@singledispatch` function, the `register` decorator is then used to define functions for each type that is handled. Dispatch to the correct function is carried out based on the type of the first argument to the function call hence the name, `single generic dispatch`. In the event that no function is defined for the type of the first argument then the base generic function, `fun` in this case is called.

Sequences and Functional Programming

Sequences such as lists and tuples play a central role in functional programming. The *Structure and Interpretation of Computer Programs*, one of the greatest computer science books ever written devotes almost a whole chapter to discussing sequences and their processing. The importance of sequences can be seen from their pervasiveness in the language. Built-ins such as `map` and `filter` consume and produce sequences. Other built-

ins such as `min`, `max`, `reduce` etc. consume sequence and return values. Functions such `range`, `dict.items()` produce sequences.

The ubiquity of sequences requires that they are represented efficiently. One could come up with multiple ways of representing sequences. For example, a naive way of implementing sequences would be to store all the members of a sequence in memory. This however has a significant drawback that sequences are limited in size to the RAM available on the machine. A more clever solution is to use a single object to represent sequences. This object knows how to compute the next required elements of the sequence on the fly just as it is needed. Python has a built-in protocol exactly for doing this, the `__iter__` protocol. This is strongly related to *generators*, a brilliant feature of the language and these are both dived into in the next chapter.

7. Iterators and Generators

In the last section of the previous chapter, the central part sequences play in functional programming and the need for their efficient representation was mentioned. The idea of representing a sequence as an objects that computes and returns the next value of a sequence just at the time such value is needed for computation was also introduces. This may seem hard to grasp at first but this chapter is dedicated to explaining all about this wonderful idea. It however begins with a description of a profound construct that has been left out of the discussion till now, *iterators*.

7.1 Iterators

An iterable in Python is any object that implements the `__iter__` special method that when called returns an *iterator* (the `__iter__` special method is invoked by a call to `iter(obj)`). Simply put, a Python iterable is any type that can be used with a `for..in` loop. Python `lists`, `tuples`, `dicts` and `sets` are all examples of built-in iterables. *Iterators* are objects that implement the *iterator protocol*. The iterator protocol in defines the following set of methods that need to be implemented by any object that wants to be used as an iterator.

- `__iter__` method that is called on initialization of an iterator. This should return an object that has a `__next__` method.
- `__next__` method that is called whenever the next() global function is invoked with the iterator as argument. The iterator's `__next__` method should return the next value of the iterable. When an iterator is used with a `for…in` loop, the `for` loop implicitly calls `next()` on the iterator object. This method should raise a `StopIteration` exception when there is no longer any new value to return to signal the end of the iteration.

Care should be taken when distinguishing between an `iterable` and an `iterator` because an iterable is not necessarily an iterator. The following snippet shows how this is possible.

```
    >>> x = [1, 2, 3]
    >>> type(x)
    <class 'list'>
    >>> x_iter = iter(x)
    >>> type(x_iter)
    <class 'list_iterator'>
    # x is iterable & can be used in a for loop but is not an iterators as it
    # does not have the __next__ method
    >>> dir(x)
    ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '
__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

```
    # x_iter is an iterator as it has the __iter__ and __next__ methods
    >>> dir(x_iter)
    ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getatt
bute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__length_hint__', '__lt__', '__ne
_', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__',
 '__sizeof__', '__str__', '__subclasshook__']
```

Something worth noting is that most times an iterable object is also an iterator so a call to such an object's `__iter__` special method returns the object itself. This will be seen later on in this section.

Any class that fully implements the *iterator protocol* can be used as an iterator. This is illustrated in the following by implementing a simple iterator that returns Fibonacci numbers up to a given maximum value.

```
class Fib:
     def __init__(self, max):
          self.max = max

     def __iter__(self):         self.a = 0      self.b = 1      return self #
     object is an iterable and an iterator

     def __next__(self):
          fib = self.a
          if fib > self.max:
               raise StopIteration
          self.a, self.b = self.b, self.a + self.b
          return fib

>>>for i in Fib(10):
          print i

0
1
1
2
3
5
8
```

A custom `range` function for looping through numbers can also be modelled as an iterator. The following is a simple implementation of a range function that loops from `0` upwards.

```
class CustomRange:
     def __init__(self, max):
          self.max = max

     def __iter__(self):
          self.curr = 0
          return self

     def __next__(self):
          numb = self.curr
          if self.curr >= self.max:
               raise StopIteration
          self.curr += 1
          return numb
```

```
for i in CustomRange(10):
      print i
0
1
2
3
4
5
6
7
8
9
```

Before attempting to move on, stop for a second and study both examples carefully. The essence of an iterator is that an iterator object knows how to calculate and return the elements in the sequence as needed not all at once. The `CustomRange` does not return all the elements in the range when it is initialized rather it returns an object that when the object's `__iter__` method is called returns an iterator object that can calculate the next element of the range using the steps defined in the `__next__` method. It is possible to define a `range` function that returns all positive whole numbers (an infinite sequence) by simply removing the upper bound on the method. The same idea applies to the `Fib` iterator. This basic idea just explained above can be seen in built-in functions that return sequences. For example, the built-in `range` function does not return a list as one would intuitively expect but returns an object that returns a `range` iterator object when its `__iter__` method is called. To get the sequence as expected the `range` iterator object is passed to the list constructor as shown in the following example.

```
>>> ran = range(0, 10)
>>> type(ran)
<class 'range'>
>>> dir(ran)
['__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__g
__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__
en__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '_
setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index', 'start', 'step', 'stop']
>>> iter = ran.__iter__()
>>> iter
<range_iterator object at 0x1012a4090>
>>> type(iter)
<class 'range_iterator'>
>>> iter.__next__()
0
>>> iter.__next__()
1
>>> iter.__next__()
2
>>> iter.__next__()
3
>>> iter.__next__()
4
>>> iter.__next__()
5
>>> iter.__next__()
6
```

```
>>> iter.__next__()
7
>>> iter.__next__()
8
>>> iter.__next__()
9
>>> iter.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> ran=range(10)
>>> ran
range(0, 10)
>>> list(ran) # use list to calculate all values in the sequence at once
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

The *iterator* protocol implements a form of computing that is referred to as *lazy computation*; it does not do more work than it has to do at any given time.

The `Itertools` Module

The concept of *iterators* is so important that Python comes with a module, the `itertools` module, that provides some useful general purpose functions that return iterators. The results of these functions can be obtained eagerly by passing the returned iterator to the `list()` constructor. A few of these functions are described below.

1. `accumulate(iterable[, func])`: This takes an *iterable* and an optional `func` argument that defaults to the `operator.add` function. The supplied function should take two arguments and return a single value. The elements of the *iterable* must be a type that is acceptable to the supplied function. A call to `accumulate` returns an iterator that represents the result of applying the supplied function to the elements of the *iterable*. The accumulated result for the first element of an iterable is the element itself while the accumulated result for the *nth* element is `func(nth element, accumulated result of (n-1)th element)`. Examples of the usage of this function are shown in the following snippet.

    ```
    >>> from itertools import *
    >>> accumulate([1,2,3,4,5])
    <itertools.accumulate object at 0x101c67c08>
    >>> list(accumulate([1,2,3,4,5]))
    [1, 3, 6, 10, 15]
    >>> import operator
    >>> accumulate(range(1, 10), operator.mul)
    <itertools.accumulate object at 0x101c6d0c8>
    >>> list(accumulate(range(1, 10), operator.mul))
    [1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
    >>>
    ```

2. `chain(*iterables)`: This takes a single iterable that contains a variable number of iterables and returns an iterator representing a union of all the iterables in supplied iterable.

```
>>> x = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
>>> chain.from_iterable(x)
<itertools.chain object at 0x101c6a208>
>>> list(chain.from_iterable(x))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
>>>
```

3. `combinations(iterable, r)` This returns an iterator representing a set of *r* length sub-sequences of elements from the input iterable. Elements are treated as unique based on their value and not on their position.

```
>>> combinations('ABCDE', 3)
<itertools.combinations object at 0x101c71138>
>>> list(combinations('ABCDE', 3))
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'B', 'E'), ('A', 'C', 'D'), ('A', 'C', 'E'),
, 'E'), ('B', 'C', 'D'), ('B', 'C', 'E'), ('B', 'D', 'E'), ('C', 'D', 'E')]
>>>
```

4. `filterfalse(predicate, iterable)`:: This returns an iterator that filters elements from the iterable argument returning only those for which the value of applying the predicate to the element is `False`. If predicate is `None`, the function returns the items that are false.
5. `groupby(iterable, key=None)`: This returns an iterator that returns consecutive keys and corresponding groups for these keys from the iterable argument. The `key` argument is a function computing a key value for each element. If a key function is not specified or is `None`, the key defaults to an identity function that returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function. The returned group is itself an iterator that shares the underlying iterable with `groupby()`. An example usage of this is shown in the following snippet.

```
>>> from itertools import groupby
>>> {k:list(g) for k, g in groupby('AAAABBBCCD')}
{'D': ['D'], 'B': ['B', 'B', 'B'], 'A': ['A', 'A', 'A', 'A'], 'C': ['C', 'C']}
>>> >>> [k for k, g in groupby('AAAABBBCCDAABBB')]
['A', 'B', 'C', 'D', 'A', 'B']
```

6. `islice(terable, start, stop[, step])`: This returns an iterator that returns elements from the iterable that are within the specified range. If start is non-zero, then elements from the iterable are skipped until start is reached. Afterwards, elements are returned consecutively with `step` elements skipped if step is greater than one just as in the conventional `slice` until the iterable argument is exhausted. Unlike conventional slicing,`islice()` does not support negative values for start, stop, or step.
7. `permutation(iterable, r=None)`: This returns a succession of `r` length permutations of elements in the iterable. If r is not specified or is None, it defaults to the length of the iterable. Elements are treated as unique based on their position, not on their value and this is where permutations differs from combinations that was previously defined. So if the input elements are unique, there will be no repeat values in each permutation.
8. `product(*iterables, repeat=1)`: This returns a iterator that returns successive Cartesian product of input iterables. This is equivalent to nested for-loops in a list

expression. For example, `product(A, B)` returns an iterator that returns values that are the same as `[(x,y) for x in A for y in B]`. This function can compute the product of an iterable with itself by specifying the number of repetitions with the optional repeat keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

7.2 Generators

Generators and iterators have a very intimate relationship. In short, Python generators are iterators and understanding generators gives one an idea of how iterators can be implemented. This may sound quite circular but after going through an explanation of generators, it will become clearer. *PEP 255* that describes simple generators refers to generators by their full name, *generator-iterators*. Generators just like the name suggests *generate* (or consume) values when their `__next__` method is called. Generators are used either by explicitly calling the `__next__` method on the generator object or using the generator object in a `for…in` loop. Generators are of two types:

1. Generator Functions
2. Generator Expressions

Generator Functions

Generator functions are functions that contain the `yield` expression. Calling a function that contains a `yield` expression returns a generator object. For example, the `Fibonacci` iterator can be recast as a generator using the `yield` keyword as shown in the following example.

```
def fib(max):
     a, b = 0, 1
     while a < max:
          yield a
          a, b = b, a + b
```

The `yield` keyword

The yield keyword has the following syntax.

```
`yield expression_list`
```

The `yield` keyword expression is central to generator functions but what does this expression really do? To understand the `yield` expression, contrast it with the `return` keyword. The `return` keyword when encountered returns control to the caller of a function effectively ending the function execution. This is shown in the following example by calling the normal *Fibonacci* function to return all Fibonacci numbers less than 10.

```
>>> def fib(max):
...      numbers = []
...      a, b = 0, 1
...      while a < max:
...           numbers.append(a)
...           a, b = b, a+b
...      return numbers
```

```
...
>>> fib(10) # all values are returned at once
[0, 1, 1, 2, 3, 5, 8]
```

On the other hand, the presence of the `yield` expression in a function complicates things a bit. When a function with a `yield` expression is called, the function does not run like a normal function rather it returns a generator expression. This is illustrated by a call to the `fib` function in the following snippet.

```
>>> f = fib(10)
>>> f
<generator object fib at 0x1013d8828>
```

The generator object executes when its `__next__` method is invoked and the generator object executes all statements in the function definition till the `yield` keyword is encountered.

```
>>> f.__next__()
0
>>> f.__next__()
1
>>> f.__next__()
1
>>> f.__next__()
2
```

The object suspends execution at that point, saves its context and returns any value in the expression_list to the caller. When the caller invokes `__next__()` method of the generator object, execution of the function continues till another `yield` or `return` expression is encountered or end of function is reached. This continues till the loop condition is false and a `StopIteration` exception is raised to signal that there is no more data to generate. To quote PEP 255,

> If a `yield` statement is encountered, the state of the function is frozen, and the value of expression_list is returned to `.__next__()'s` caller. By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `.next()` is invoked, the function can proceed exactly as if the yield statement were just another external call. On the other hand when a function encounters a `return` statement, it returns to the caller along with any value proceeding the return statement and the execution of such function is complete for all intent and purposes. One can think of `yield` as causing only a temporary interruption in the executions of a function.

With a better understanding of generators, it is not difficult to see how generators can be used to implement iterators. Generators know how to calculate the next value in a sequence so functions that return iterators can be rewritten using the `yield` statement. To illustrate this, the `accumulator` function from the `itertools` module can be rewritten using generators as in the following snippet.

```
def accumulate(iterable, func=operator.add):
     'Return running totals'
     # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
     # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
     it = iter(iterable)
     try:
          total = next(it)
     except StopIteration:
          return
     yield total
     for element in it:
          total = func(total, element)
          yield total
```

Similarly, one can emulate a generator object by implementing the *iterator* protocol discussed at the start of this chapter. However, the `yield` keyword provides a more succinct and elegant method for creating generators.

Generator Expressions

In the previous chapter, list comprehensions were discussed. One drawback with list comprehensions is that values are calculated all at once regardless of whether the values are needed at that time or not (eager calculation). This may sometimes consume an inordinate amount of computer memory. PEP 289 proposed the generator expression to resolve this and this proposal was accepted and added to the language. Generator expressions are like list comprehensions; the only difference is that the square brackets in list comprehensions are replaced by circular brackets that return a *generator expression object*.

To generate a list of the square of number from 0 to 10 using list comprehensions, the following is done.

```
>>> squares = [i**2 for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A generator expression could be used in place of a list comprehension as shown in the following snippet.

```
>>> squares = (i**2 for i in range(10))
>>> squares
<generator object <genexpr> at 0x1069a6d70>
```

The values of the generator can then be accessed using `for…in` loops or via a call to the `__next__()` method of the generator object as shown below.

```
>>> squares = (i**2 for i in range(10))
>>> for square in squares:
                print(square)
0
1
4
9
16
```

```
25
36
49
64
81
```

Generator expression create generator objects without using the `yield` expression.

The Beauty of Generators and Iterators

Generators really shine when working with massive amounts of data streams. Consider the very representative but rather trivial example of generating a stream of *prime numbers*. A method for calculating this set is the *Sieve of Eratosthenes*. The following algorithm will find all the prime numbers less than or equal to a given integer, n, by the sieve of Eratosthenes' method:

```
    1. Create a list of consecutive integers from 2 to n: (2, 3, 4, ..., n).
    2. Initially, let p equal 2, the first prime number.
    3. Starting from p, enumerate its multiples by counting to n in increments of p, and mark th
n the list (these will be 2p, 3p, 4p, ... ; p itself should not be marked).
    4. Find the first number greater than p in the list that is not marked. If there was no such
ber, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from
p 3.
```

When the algorithm terminates, the remaining numbers not marked in the list are all the primes below n. Now this is a rather trivial algorithm and this is implemented using generators.

```
from itertools import count

def primes_to(gen, max_val):
     for i in range(max_val):
          print(gen.__next__())

def filter_multiples_of_n(n, ints):
     # ints is a generator that can have other generators within it
     for i in ints:
          if (i % n) != 0:
               yield i

def sieve(ints):
     while True:
          prime = ints.__next__()
          yield prime
          # ints is now a generator that produces integers that are not
          # multiples of prime
          ints = filter_multiples_of_n(prime, ints)

# all prime numbers less than 400
primes_to(sieve(count(2)), 10)
```

```
2
3
5
7
11
13
```

```
17
19
23
29
```

The above example though very simple, shows the beauty of how generators can be chained together with the output of one acting as input to another; think of this stacking of generators with one another as a kind of processing pipeline. The `filter_multiples_of_n` function is worth discussing a bit here because it maybe confusing at first. `counts(2)` when initialized returns a generator that returns a sequence of consecutive numbers from 2 so the line, `prime=ints.__next__()` returns `2` on the first iteration. After the `yield` expression, `ints=filter_multiples_of_n(2, ints)` is invoked creating a generator that returns a stream of numbers that are not multiples of `2` - note that the original sequence generator is captured within this new generator (this is very important). Now on the next iteration of the `loop` within the `sieve` function, the `ints` generator is invoked. The generator loops through the original sequence now `[3, 4, 5, 6, 7, ....]` yielding the first number that is not divisible by 2, 3 in this case. This part of the pipeline is easy to understand. The prime, `3`, is yielded from the `sieve` function then another generator that returns non-multiples of the prime, 3, is created and assigned to `ints`. This generator captures the previous generator that produces non- multiples of `2` and that generator captured the original generator that produces sequences of infinite consecutive numbers. A call to the `__next__()` method of this generator will loop through the previous generator that returns non-multiples of `2` and every non-multiple of `2` returned by the generator is checked for divisibility by `3` and if the number is not divisible by `3` it is yielded. This chaining of generators goes on and on. The next prime is `5` so the generator excluding the multiples of primes will loop through the generator that returns non-multiples of 3 which in turn loops through the generator that produces non-multiple of 2.

This streaming of data through multiple generators can be applied to the space and sometime time efficient processing of any other stream of massive data such as log files and data bases. Generators however have other nifty and mind-blowing use cases as will be seen in the following sections.

7.3 From Generators To Coroutines

> “Subroutines are special cases of … coroutines.”
>
> – Donald Knuth

A subroutine is a set of program instructions bundled together to perform a specific task. Functions and methods are examples of subroutines in Python. Subroutines have a single point of entry or exit; this is seen in ordinary functions and methods which once called execute till they exit and cannot be suspended. Coroutines however are a more general program construct that allow multiple entry points for suspending and resuming execution. Multiple entry points for suspending and resuming sounds exactly just like what the `yield` expression provides to generator functions and in-fact one could argue that Python generators are in-fact

coroutines because they allow the production and consumption of values at their suspension or resumption points. The `send()` method of generators added in Python version 2.5 provides generators with the ability to consume data when a generator resumes execution. The documentation provided for the `send()` method by the Python documentation follows.

> *generator.send(value)*: Resumes the execution and "sends" a value into the generator function. The value argument becomes the result of the current yield expression. The send() method returns the next value yielded by the generator, or raises StopIteration if the generator exits without yielding another value. When send() is called to start the generator, it must be called with None as the argument, because there is no yield expression that could receive the value.

The above explanation maybe a little ambiguous so an illustration of the use of the `send()` method is provided with the following example.

```
>>> def find_pattern(pattern):
...       print("looking for {}".format(pattern))
...       while True:
...            line = yield
...            if pattern in line:
...                 print(line)
...
```

The generator is initialized and run as shown in the following snippet.

```
>>> g = find_pattern('python')
# coroutines must be primed with a call to next before sending in values
>>> g.__next__()
looking for python
>>> g.send("Yeah, but no, but year, but no")
>>> g.send("python generators rock!")
python generators rock!
```

To fully grasp the `send()` method, observe that the argument passed to the `send()` method of the generator will be the result of the `yield` expression so in the above example, the value that `send()` is called with is assigned to the variable, `line`. The rest of the function is straightforward to understand. Note that calling `send(None)` is equivalent to calling the generator's `__next__()` method.

Simulating Multitasking with Coroutines

The ability to send data into generators using the `send()` method truly expands the frontier for generators. Now think carefully about the tools in our arsenal at this point:

1. Functions that can run independent of one another while maintaining state
2. Functions that can suspend execution and resume execution multiple times.
3. Functions that can receive data at resumption.

A little thinking shows that multiple generators or coroutines can be scheduled to run in an interleaved manner and it would be like they were executing simultaneously. With that, we

have multitasking or something like it. In this section, rudimentary multitasking is simulated to illustrate the versatility of generators/coroutines.

In reality, even a full blown multitasking operating system is only ever executing a single task at one time. A task is any set of instructions with own internal state. For example, a simple task may take a stream of text count the occurrence of some words and print a running count of some words in the stream or may just print any word it receives. What is important here is that tasks are totally independent of each other. The illusion of multitasking is achieved by giving each task a slice of time to run until it encounters a *trap* which forces it to stop running so that other tasks can run. This happens so fast that human users cannot sense what is happening. This can easily be simulated with a collection of coroutines that run independent of each as shown in this section. A very simple example of this multitasking is shown in the following snippet in which text is read from a source and then sent for processing to multiple coroutines. In the snippet, a task is modelled as a thin wrapper around a coroutine.

```
from collections import defaultdict

class Task():

     def __init__(self, coroutine):
          self.coroutine = coroutine
          next(self.coroutine)

     def run(self, value):
          self.coroutine.send(value)

def read(source):
     for line in source:
          yield line

def print_line():
     while True:
          text = yield
          print(text)

def word_count():
     word_counts = defaultdict(int)
     while True:
          text = yield
          for word in text.split():
               word_counts[word] += 1
          print("Word distribution so far is ", word_counts)


def run():
     f = open("data.txt")
     source = read(f)
     tasks = [ Task(print_line()), Task(word_count()) ]
     for line in source:
          for task in tasks:
               try:
                    task.run(line)
               except StopIteration:
                    tasks.remove(task)
```

```
    if __name__ == '__main__':
        run()

    We love python don't we?
    Word distribution so far is  defaultdict(<class 'int'>, {"don't": 1, 'we?': 1, 'python': 1, 'l
e': 1, 'We': 1})
    No we don't love python
    Word distribution so far is  defaultdict(<class 'int'>, {"don't": 2, 'we?': 1, 'python': 2, 'w
: 1, 'We': 1, 'No': 1, 'love': 2})
```

Observer how the outputs are interleaved because execution of each coroutine happens for a limited time then another coroutines executes. The above example is very instructive in showing the power of generators and coroutines. The above has just been provided for illustration purposes. The `asyncio` module is provided in Python 3.5 for concurrent programming using coroutines.

7.4 The `yield from` keyword

Sometimes re-factoring a code block may involve moving some functionality into another generator. Using just the `yield` keyword may prove cumbersome in some of these cases and impossible in other cases such as when there is a need to send data to the delegated generator that was sent to the delegating generator. This kind of scenarios led to the introduction of the `yield from` keyword.
This keyword allows a section of code containing `yield` to be moved into another generator. Furthermore, the delegated generator can return a value that is made available to the delegating generator. An instructive example on how the `yield from` keyword works is given in the following example (note that a call to the `range` function returns a generator).

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

As previously mentioned, *yielding* data from a delegated generator was not the only reason for the introduction of the `yield from` keyword because the previous `yield from` snippet can be replicated without `yield from` as shown in the following example.

```
>>> def g(x):
...     r = []
...     for i in range(x, 0, -1):
...         r.append(i)
...     for j in range(x):
...         r.append(j)
...     return r
...
>>> x = g(5)
>>> x
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

The real benefit of using the new `yield from` keyword comes from the ability of a calling generator to *send* values into the delegated generator as shown in the following example. Thus if a value is sent into a generator `yield from` enables that generator to also implicitly send the same value into the delegated generator.

```
>>> def accumulate():
...     tally = 0
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc) # Ensure the accumulator is ready to accept values
>>> for i in range(4):
...     acc.send(i)
...
>>> acc.send(None) # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None) # Finish the second tally
>>> tallies
[6, 10]
```

The complete semantics for `yield from` is explained in PEP 380 and given below.

1. Any values that the iterator yields are passed directly to the caller.
2. Any values sent to the delegating generator using `send()` are passed directly to the iterator. If the sent value is `None`, the iterator's `__next__()` method is called. If the sent value is not `None`, the iterator's `send() method is called. If the call raises StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
3. Exceptions other than `GeneratorExit` thrown into the delegating generator are passed to the `throw()` method of the iterator. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
4. If a `GeneratorExit` exception is thrown into the delegating generator, or the `close()` method of the delegating generator is called, then the `close()` method of the iterator is called if it has one. If this call results in an exception, it is propagated to the delegating generator. Otherwise, `GeneratorExit` is raised in the delegating generator.
5. The value of the yield from expression is the first argument to the `StopIteration` exception raised by the iterator when it terminates.
6. `return expr` in a generator causes `StopIteration(expr)` to be raised upon exit from the generator.

7.5 A Game of Life

To end the chapter, a very simple game, *Conway's Game of Life*, is implemented using generators and coroutines to simulate the basics of the game; a thorough understanding of this example will prove further enlightening. This example is inspired by Brett Slatkin's *Effective Python* chapter on using coroutines for running thousands of function and all credits are due to him.

An explanation of the Game of Life as given by Wikipedia follows. The Game of Life is a simulation that takes place on a two-dimensional orthogonal grid of cells each of which is either in an alive or dead state. Each cell transitions to a new state in a step of time by its interaction with its neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step of time, the following transitions occur:

```
1. Any live cell with fewer than two live neighbours dies.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies.
4. Any dead cell with exactly three live neighbours becomes a live cell.
```

The initial pattern of cells on the grid constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell and the discrete moment at which this happens is sometimes called a tick. The rules continue to be applied repeatedly to create further generations.

In the following implementation, each cell's simulation is carried out using a coroutine with the state of the cells stored in a `Grid` object from generation to generation.

```
    # Copyright 2014 Brett Slatkin, Pearson Education Inc.
    #
    # Licensed under the Apache License, Version 2.0 (the "License");
    # you may not use this file except in compliance with the License.
    # You may obtain a copy of the License at
    #
    #     http://www.apache.org/licenses/LICENSE-2.0
    #
    # Unless required by applicable law or agreed to in writing, software
    # distributed under the License is distributed on an "AS IS" BASIS,
    # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    # See the License for the specific language governing permissions and
    # limitations under the License.

    from collections import namedtuple

    ALIVE = '*'
    EMPTY = '-'
    TICK = object()

    Position = namedtuple('Position', 'y x')

    Transition = namedtuple('Transition', 'y x state')

    def count_neighbors(y, x):
        n_ = yield Position(y + 1, x + 0)  # North
        ne = yield Position(y + 1, x + 1)  # Northeast
        e_ = yield Position(y + 0, x + 1)  # East
```

```
    se = yield Position(y - 1, x + 1)  # Southeast
    s_ = yield Position(y - 1, x + 0)  # South
    sw = yield Position(y - 1, x - 1)  # Southwest
    w_ = yield Position(y + 0, x - 1)  # West
    nw = yield Position(y + 1, x - 1)  # Northwest
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    return sum([1 for state in neighbor_states if state is == ALIVE])

def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2 or neighbors > 3:
            return EMPTY      # Die: Too few
    else:
        if neighbors == 3:
            return ALIVE      # Regenerate
    return state

def transition_cell(y, x):
    # request info on the state of the cell at y, x
    state = yield Position(y, x)
    neighbors = yield from count_neighbors(y, x)
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)

def simulate(height, width):
    while True:
        for y in range(height):
            for x in range(width):
                yield from transition_cell(y, x)
        yield TICK

def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Position):
            state = grid[item.y, item.x]
            item = sim.send(state)
        else:  # Must be a Transition
            progeny[item.y, item.x] = item.state
            item = next(sim)
    return progeny

class Grid(object):
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def __str__(self):
        output = ''
        for row in self.rows:
            for cell in row:
                output += cell
            output += '\n'
        return output
```

```
        def __getitem__(self, position):
            y, x = position
            return self.rows[y % self.height][x % self.width]

        def __setitem__(self, position, state):
            y, x = position
            self.rows[y % self.height][x % self.width] = state

    class ColumnPrinter(object):
        def __init__(self):
            self.columns = []

        def append(self, data):
            self.columns.append(data)

        def __str__(self):
            row_count = 1
            for data in self.columns:
                row_count = max(row_count, len(data.splitlines()) + 1)
            rows = [''] * row_count
            for j in range(row_count):
                for i, data in enumerate(self.columns):
                    line = data.splitlines()[max(0, j - 1)]
                    if j == 0:
                        rows[j] += str(i).center(len(line))
                    else:
                        rows[j] += line
                    if (i + 1) < len(self.columns):
                        rows[j] += ' | '
            return '\n'.join(rows)

    grid = Grid(5, 5)
    grid[1, 1] = ALIVE
    grid[2, 2] = ALIVE
    grid[2, 3] = ALIVE
    grid[3, 3] = ALIVE
    columns = ColumnPrinter()
    sim = simulate(grid.height, grid.width)
    for i in range(6):
        columns.append(str(grid))
        grid = live_a_generation(grid, sim)
    print(columns)
```

```
  0   |   1   |   2   |   3   |   4   |   5
----- | ----- | ----- | ----- | ----- | -----
-*--- | --*-- | --**- | --*-- | ----- | -----
--**- | --**- | -*--- | -*--- | -**-- | -----
---*- | --**- | --**- | --*-- | ----- | -----
----- | ----- | ----- | ----- | ----- | -----
```

Generators are a fascinating topic and this chapter has barely scratched the surface of what is possible. David Beazley gave a series of excellent talks, 1,2 and 3, that go into great detail about very advanced usage of generators.

8. MetaProgramming and Co.

Metaprogramming is quite an interesting area of programming. Metaprogramming deals with code that manipulates other code. It is a broad category that covers areas such as function decorators, class decorators, metaclasses and the use of built-ins like `exec`, `eval` and context managers etc. These constructs sometimes help to prevent repetitive code and most times add new functionality to a piece of code in elegant ways. In this chapter, decorators, metaclasses and context managers are discussed.

8.1 Decorators

A decorator is a function that wraps another function or class. It introduces new functionality to the wrapped class or function without altering the original functionality of such class or function thus the interface of such class or function remains the same.

Function Decorators

A good understanding of functions as first class objects is important in order to understand *function decorators*. A reader will be well served by reviewing the material on functions. When functions are first class objects the following will apply to functions:

1. Functions can be passed as arguments to other functions.
2. Functions can be returned from other function calls.
3. Functions can be defined within other functions resulting in closures.

The above listed properties of first class functions provide the foundation needed to explain ***function decorators***. Put simply, function decorators are ***"wrappers" that enable the execution of code before and after the function they decorate without modifying the function itself***.

Function decorators are not unique to Python so to explain them, Python function decorators and the corresponding syntax are ignored for the moment and instead the essence of function decorators is focused on. To understand what decorators do, a very trivial function is decorated with another trivial function that logs calls to the decorated function in the following example. This function decoration is achieved using function composition as shown below (follow the comments):

```
import datetime

# decorator expects another function as argument
def logger(func_to_dec):

    # A wrapper function is defined on the fly
    def func_wrapper():

        # add any pre original function execution functionality
        print("Calling function: {} at {}".format(func_to_dec.__name__, datetime.datetime.now(
```

```
            # execute original function
            func_to_dec()

            # add any post original function execution functionality
            print("Finished calling : {}".format(func_to_dec.__name__))

        # return the wrapper function defined on the fly. Body of the
        # wrapper function has not been executed yet but a closure
        # over the func_to_decorate has been created.
        return func_wrapper

def print_full_name():
    print("My name is John Doe")

# use composition to decorate the print_full_name function
>>>decorated_func = logger(print_full_name)
>>>decorated_func
# the returned value, decorated_func, is a reference to a func_wrapper
<function func_wrapper at 0x101ed2578>
>>>decorated_func()
# decorated_func call output
Calling function: print_full_name at 2015-01-24 13:48:05.261413
# the original functionality is preserved
My name is John Doe
Finished calling : print_full_name
```

In the trivial example defined above, the decorator adds a new feature, printing some information before and after the original function call, to the original function without altering it. The decorator, `logger` takes a function to be decorated, `print_full_name` and returns a function, `func_wrapper` that calls the decorated function, `print_full_name`, when it is executed. The decoration process here is calling the decorator with the function to be decorated as argument. The function returned, `func_wrapper` is closed over the reference to the decorated function, `print_full_name` and thus can invoke the decorated function when it is executing. In the above, calling `decorated_func` results in `print_full_name` being executed in addition to some other code snippets that implement new functionality. This ability to add new functionality to a function without modifying the original function is the essence of function decorators. Once this concept is understood, the concept of decorators is understood.

Decorators in Python

Now that the essence of function decorators have been discussed, an attempt is made to de-construct Python constructs that enable the definition of decorators more easily. The previous section describes the essence of decorators but having to use decorators via function compositions as described is cumbersome. Python introduces the `@` symbol for decorating functions. Decorating a function using the Python decorator syntax is achieved as shown in the following example.

```
@decorator
def a_stand_alone_function():
    pass
```

Calling `stand_alone_function` now is equivalent to calling `decorated_func` function from the previous section but there is no longer a need to to define the intermediate `decorated_func`.

It is important to understand what the `@` symbol does with respect to decorators in Python. The `@decorator` line does not define a python decorator rather one can think of it as syntactic sugar for **decorating a function**. I like to define **decorating a function** as the process of applying an existing decorator to a function. The ***decorator*** is the actual function, `decorator`, that adds the new functionality to the original function. According to PEP 318, the following decorator snippet

```
@dec2
@dec1
def func(arg1, arg2, ...):
     pass
```

is equivalent to

```
def func(arg1, arg2, ...):
     pass

func = dec2(dec1(func))
```

without the intermediate `func` argument. In the above, `@dec1` and `@dec2` are the decorator invocations. Stop, think carefully and ensure you understand this. `dec1` and `dec2` are function object references and these are the actual decorators. These values can even be replaced by any ***function call or a value that when evaluated returns a function that takes another function***. What is of paramount importance is that the *name reference* following the `@` symbol is a reference to a function object (for this tutorial we assume this should be a function object but in reality it should be a ***callable*** object) that takes a function as argument. Understanding this profound fact will help in understanding python decorators and more involved decorator topics such as decorators that take arguments.

Passing Arguments To Decorated Functions

Arguments are supplied to functions that are being decorated by simply passing the arguments into the function that wraps, **i.e the inner function returned when the decorator is invoked**, the decorated function. This is illustrated with the following example.

```
import datetime

# decorator expects another function as argument
def logger(func_to_decorate):

     # A wrapper function is defined on the fly
     def func_wrapper(*args, **kwargs):

          # add any pre original function execution functionality
          print("Calling function: {} at {}".format(func_to_decorate.__name__, datetime.datetime
ow()))

          # execute original function
```

```
            func_to_decorate(*args, **kwargs)

            # add any post original function execution functionality
            print("Finished calling : {}".format(func_to_decorate.__name__))

        # return the wrapper function defined on the fly. Body of the
        # wrapper function has not been executed yet but a closure over
        # the func_to_decorate has been created.
        return func_wrapper

    @logger
    def print_full_name(first_name, last_name):
        print("My name is {} {}".format(first_name, last_name))

    print_full_name("John", "Doe")

    Calling function: print_full_name at 2015-01-24 14:36:36.691557
    My name is John Doe
    Finished calling : print_full_name
```

Note how the `*args` and `**kwargs` parameters are used in defining the inner wrapper function; this is for the simple reason that it cannot be known beforehand what functions are going to be decorated and thus the function signature of such functions.

Decorator Functions with Arguments

Decorator functions can also be defined to take arguments but this is more involved than the case of passing functions to decorated functions. The following example illustrates this.

```
    # this function takes arguments and returns a function.
    # the returned functions is our actual decorator
    def decorator_maker_with_arguments(decorator_arg1):

        # this is our actual decorator that accepts a function

        def decorator(func_to_decorate):

            # wrapper function takes arguments for the decorated
            # function
            def wrapped(function_arg1, function_arg2) :
                # add any pre original function execution
                # functionality
                print("Calling function: {} at {} with decorator arguments: {} and function argum
s:{} {}".\
                      format(func_to_decorate.__name__, datetime.datetime.now(), decorator_arg1, 
ion_arg1, function_arg2))

                func_to_decorate(function_arg1, function_arg2)

                # add any post original function execution
                # functionality
                print("Finished calling : {}".format(func_to_decorate.__name__))

            return wrapped

        return decorator
```

```
@decorator_maker_with_arguments("Apollo 11 Landing")
def print_name(function_arg1, function_arg2):
    print ("My full name is—{} {} --".format(function_arg1, function_arg2))

>>> print_name("Tranquility base ", "To Houston")

Calling function: print_name at 2015-01-24 15:03:23.696982 with decorator arguments: Apollo 11
anding and function arguments:Tranquility base  To Houston
My full name is -- Tranquility base  To Houston --
Finished calling : print_name
```

As mentioned previously, the key to understanding what is going on with this is to note that we can replace the reference value following the `@` in a function decoration with any value that ***evaluates to a function object that takes another function as argument***. In the above snippet, the value returned by the function call, `decorator_maker_with_arguments("Apollo 11 Landing")` , is the decorator. The call evaluates to a function, `decorator` that accepts a function as argument. Thus the decoration `@decorator_maker_with_arguments("Apollo 11 Landing")` is equivalent to `@decorator` but with the decorator, `decorator` , closed over the argument, `Apollo 11 Landing` by the `decorator_maker_with_arguments` function call. Note that the arguments supplied to a decorator can not be dynamically changed at run time as they are executed on script import.

Functools.wrap

Using decorators involves swapping out one function for another. A result of this is that *meta* information such as docstrings in the swapped out function are lost when using a decorator with such function. This is illustrated below:

```
import datetime

# decorator expects another function as argument
def logger(func_to_decorate):

    # A wrapper function is defined on the fly
    def func_wrapper():

        # add any pre original function execution functionality
        print("Calling function: {} at {}".format(func_to_decorate.__name__, datetime.datetime
ow()))

        # execute original function
        func_to_decorate()

        # add any post original function execution functionality
        print("Finished calling : {}".format(func_to_decorate.__name__))

    # return the wrapper function defined on the fly. Body of the
    # wrapper function has not been executed yet but a closure
    # over the func_to_decorate has been created.
    return func_wrapper

@logger
def print_full_name():
```

```
        """return john doe's full name"""
        print("My name is John Doe")

>>> print(print_full_name.__doc__)
None
>>> print(print_full_name.__name__)
func_wrapper
```

In the above example, an attempt to print the documentation string returns `None` because the decorator has swapped out the `print_full_name` function with the `func_wrapper` function that has no documentation string. Even the function name now references the name of the wrapper function rather than the actual function. This, most times, is not what we want when using decorators. To work around this Python `functools` module provides the `wraps` function that also happens to be a decorator. This decorator is applied to the wrapper function and takes the function to be decorated as argument. The usage is illustrated in the following example.

```
import datetime
from functools import wraps

# decorator expects another function as argument
def logger(func_to_decorate):

    @wraps(func_to_decorate)
    # A wrapper function is defined on the fly
    def func_wrapper(*args, **kwargs):

        # add any pre original function execution functionality
        print("Calling function: {} at {}".format(func_to_decorate.__name__, datetime.datetime
ow()))

        # execute original function
        func_to_decorate(*args, **kwargs)

        # add any post original function execution functionality
        print("Finished calling : {}".format(func_to_decorate.__name__))

    # return the wrapper function defined on the fly. Body of the
    # wrapper function has not been executed yet but a closure over
    # the func_to_decorate has been created.
    return func_wrapper

@logger
def print_full_name(first_name, last_name):
    """return john doe's full name"""
    print("My name is {} {}".format(first_name, last_name))

>>> print(print_full_name.__doc__)
return john doe's full name
>>>print(print_full_name.__name__)
print_full_name
```

Class Decorators

Like functions, classes can also be decorated. Class decorations server the same purpose as function decorators - introducing new functionality without modifying the actual

classes. An example of a class decorator is given in the following *singleton* decorator that ensures that only one instance of a decorated class is ever initialised throughout the lifetime of the execution of the program.

```
def singleton(cls):
     instances = {}

     def get_instance():
          if cls not in instances:
               instances[cls] = cls()
          return instances[cls]
     return get_instance
```

Putting the decorator to use in the following examples shows how this works. In the following example, the `Foo` class is initialized twice however comparing the *ids* of both initialized objects shows that they both refer to the same object.

```
@singleton
class Foo(object):
     pass

>>> x = Foo()
>>> id(x)
4310648144
>>> y = Foo()
>>> id(y)
4310648144
>>> id(y) == id(x) # both x and y are the same object
True
>>>
```

The same singleton functionality can be achieved using a metaclass by overriding the `__call__` method of the metaclass as shown below:

```
class Singleton(type):
     _instances = {}
     def __call__(cls, *args, **kwargs):
          if cls not in cls._instances:
               cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
          return cls._instances[cls]

class Foo(object):
     __metaclass__ = Singleton

>>> x = Foo()
>>> y = Foo()
>>> id(x)
4310648400
>>> id(y)
4310648400
>>> id(y) == id(x)
True
```

Applying Decorators to instance and static methods

Instance, static and class methods can also be decorated. The important thing is to take note of the order in which the decroators are placed in static and class methods. The decorator must come before the static and class method decorators that are used to create static and class methods because these method decorators do not return callable objects. A valid example of method decorators is shown in the following example.

```
def timethis(func):

     @wraps(func)
     def wrapper(*args, **kwargs):
          start = time.time()
          r = func(*args, **kwargs)
          end = time.time()
          print(end - start)
          return r

     return wrapper


# Class illustrating application of the decorator to different kinds of methods
class Spam:

     @timethis
     def instance_method(self, n):
          print(self, n)
          while n > 0:
               n -= 1

     @classmethod
     @timethis
     def class_method(cls, n):
          while n > 0:
               print(n)
               n -= 1

     @staticmethod
     @timethis
     def static_method(n):
          while n > 0:
               print(n)
               n -= 1


>>>Spam.class_method(10)
10
9
8
7
6
5
4
3
2
1
0.00019788742065429688
>>>Spam.static_method(10)
10
9
```

```
8
7
6
5
4
3
2
1
0.00014591217041015625
```

8.2 Decorator Recipes

Decorators have a wide range of applications in python; this section discusses some interesting uses of decorators providing the implementation for such decorators. The following are just samples of the possible applications of decorators. A more comprehensive listing of recipes including the examples listed that are discussed in this section can be found at Python decorator library website. A major benefit of a lot of decorators is that cross cutting concerns such as logging information can be done in a single place, the decorator, rather across multiple functions. The benefit of having such functionality in one place is glaringly obvious as changes are localised and way easier to maintain. The following recipes illustrate this.

1. Decorators provide a mean to log information about other functions; these maybe information such as timing information or argument information. An example of such a decorator is shown in the following example.

```
import logging

def log(func):
     '''Returns a wrapper that wraps func. The wrapper will log the entry and exit points
function with logging.INFO level.'''
          logging.basicConfig()
          logger = logging.getLogger(func.__module__)
          @functools.wraps(func)
          def wrapper(*args, **kwds):
               logger.info("About to execute {}".format(func.__name__))
               f_result = func(*args, **kwds)
               logger.info("Finished the execution of {}".format(func.__name__))
               return f_result
          return wrapper
```

2. A *memoization* decorator can be used to decorate a function that performs a calculation so that for a given argument if the result has been previously computed, the stored value is returned but if it has not then it is computed and stored before it is returned to the caller. This kind of decorator is available in the functools module as discussed in the chapter on functions. An implementation for such a decorator is shown in the following example.

```
import collections

def cache(func):
     cache = {}

     logging.basicConfig()
```

```
            logger = logging.getLogger(func.__module__)
            logger.setLevel(10)

            @functools.wraps(func)
            def wrapper(*arg, **kwds):
                 if not isinstance(arg, collections.Hashable):
                      logger.info("Argument cannot be cached: {}".format(arg))
                      return func(*arg, **kwds)
                 if arg in cache:
                      logger.info("Found precomputed result, {}, for argument, {}".format(cache[a
))
                      return cache[arg]
                 else:
                      logger.info("No precomputed result was found for argument, {}".format(arg)
                      value = func(*arg, **kwds)
                      cache[arg] = value
                      return value
            return wrapper
```

3. Decorators could also easily be used to implement functionality that retries a callable up to a maximum amount of times.

```
      def retries(max_tries, delay=1, backoff=2, exceptions=(Exception,), hook=None):
           """Function decorator implementing retrying logic. The decorator will call the functi
to max_tries times if it raises an exception.
          """
            def dec(func):
                 def f2(*args, **kwargs):
                      mydelay = delay
                      tries = range(max_tries)
                      tries.reverse()
                      for tries_remaining in tries:
                           try:
                               return func(*args, **kwargs)
                           except exceptions as e:
                                if tries_remaining > 0:
                                     if hook is not None:
                                          # hook is any function we want to call
                                          # when the original function fails
                                          hook(tries_remaining, e, mydelay)
                                     sleep(mydelay)
                                     mydelay = mydelay * backoff
                                else:
                                     raise
                           else:
                                break
                 return f2
            return dec
```

4. Another very interesting decorator recipe is the use of decorators to enforce types for function call as shown in the following example.

```
 import sys

 def accepts(*types, **kw):
 '''Function decorator. Checks decorated function's arguments are
 of the expected types.
```

```
    Parameters:
    types—The expected types of the inputs to the decorated function.
            Must specify type for each parameter.
    kw   —Optional specification of 'debug' level (this is the only valid
            keyword argument, no other should be given).
            debug = ( 0 | 1 | 2 )

    '''
    if not kw:
         # default level: MEDIUM
         debug = 1
    else:
         debug = kw['debug']
    try:
         def decorator(f):
              def newf(*args):
                   if debug is 0:
                        return f(*args)
                   assert len(args) == len(types)
                   argtypes = tuple(map(type, args))
                   if argtypes != types:
                        msg = info(f.__name__, types, argtypes, 0)
                        if debug is 1:
                             raise TypeError(msg)
                   return f(*args)
              newf.__name__ = f.__name__
              return newf
         return decorator
    except KeyError as err:
         raise KeyError(key + "is not a valid keyword argument")
    except TypeError(msg):
         raise TypeError(msg)

    def info(fname, expected, actual, flag):
         '''Convenience function returns nicely formatted error/warning msg.'''

         format = lambda types: ', '.join([str(t).split("'")[1] for t in types])
         expected, actual = format(expected), format(actual)
         msg = "'{}' method ".format( fname )\
                + ("accepts", "returns")[flag] + " ({}), but ".format(expected)\
                + ("was given", "result is")[flag] + " ({})".format(actual)
         return msg

    >>> @test_concat.accepts(int, int, int)
    ... def div_sum_by_two(x, y, z):
    ...      return sum([x, y, z])/2
    ...
    >>> div_sum_by_two('obi', 'nkem', 'chuks') # calling with wrong arguments
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/c4obi/src/test_concat.py", line 104, in newf
         raise TypeError(msg)
    TypeError: 'div_sum_by_two' method accepts (int, int, int), but was given (str, str, str)
```

5. A common use of class decorators is for registering classes as the class statements are executed as shown in the following example.

```
registry   =   {}

def register(cls):
     registry[cls.__clsid__] = cls
     return cls

@register
class Foo(object):
     __clsid__ = ".mp3"

def bar(self):
     pass
```

A more comprehensive listing of recipes including the examples listed that are discussed in this section can be found at Python decorator library website.

8.3 Metaclasses

> "Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't"
>
> – Tim Peters

All values in Python are objects including classes so a given class object must have another class from which it is created. Consider, an instance, `f`, of a user defined class `Foo`. The type/class of the instance, `f`, can be found by using the built-in method, `type` and in the case of the object, `f`,the type of `f` is `Foo`.

```
>>> class Foo(object):
...       pass
...
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
>>>
```

This introspection can also extended to a *class object* to find out the type/class of such a class. The following example shows the result of applying the `type()` function to the the `Foo` class.

```
class Foo(object):
     pass

>>> type(Foo)
<class 'type'>
```

In Python, the class of all other class objects is the `type` class. This applies to user defined classes as shown above as well as built-in classes as shown in the following code example.

```
>>>type(dict)
<class 'type'>
```

A class such as the `type` class that is used to create other classes is called a ***metaclass***. That is all there is to a metaclass - a metaclass is a class that are used in creating other classes. Custom metaclasses are not used often in Python but sometimes it is necessary to control the way classes are created most especially when working on big projects with big team.

Before explaining how metaclasses are used to customize class creation, a recap of how class objects are created when a *class* statement is encountered during the execution of a program is provided.

The following snippet is the class definition for a simple class that every Python user is familiar with but this is not the only way a class can be defined.

```
# class definition
class Foo(object):
     def __init__(self, name):
          self.name = name

     def print_name():
          print(self.name)
```

The following snippet shows a more involved method for defining the same class with all the syntactic sugar provided by the `class` keyword stripped away. This snippet provides a better understanding of what actually goes on under the covers during the execution of a class statement.

```
class_name = "Foo"
class_parents = (object,)
class_body = """
def __init__(self, name):
    self.name = name

def print_name(self):
    print(self.name)
"""
# a new dict is used as local namespace
class_dict = {}

#the body of the class is executed using dict from above as local
# namespace
exec(class_body, globals(), class_dict)

# viewing the class dict reveals the name bindings from class body
>>>class_dict
{'__init__': <function __init__ at 0x10066f8c8>, 'print_name': <function blah at 0x10066fa60>}
# final step of class creation
Foo = type(class_name, class_parents, class_dict)
```

During the execution of class statement, the interpreter carries out the following procedures behind the scene:

1. The body of the class statement is isolated in a string.
2. A class dictionary representing the namespace for the class is created.
3. The body of the class is executed as a set of statements within this namespace.

4. As a final step in the process, the class object is created by instantiating the `type` class passing in the class name, base classes and class dictionary as arguments. The `type` class used here in creating the `Account` class object is the metaclass. The metaclass value used in creating the class object can be explicitly specified by supplying the `metaclass` keyword argument in the `class` definition. In the case that it is not supplied, the class statement examines the first entry in the *tuple* of the the base classes if any. If no base classes are used, the global variable `__metaclass__` is searched for and if no value is found for this, the default metaclass value is used.

Armed with a basic understanding of metaclasses, a discussion of their applications follows.

Metaclasses in Action

It is possible to define custom metaclasses that can be used when creating classes. These custom metaclasses will normally inherit from `type` and re-implement certain methods such as the `__init__` or `__new__` methods.

Imagine that you are the chief architect for a shiny new project and you have diligently read dozens of software engineering books and style guides that have hammered on the importance of *docstrings* so you want to enforce the requirement that all non-private methods in the project must have *docstrings; how would you enforce this requirement?

A simple and straightforward solution is to create a custom metaclass for use across the project that enforces this requirement. The snippet that follows though not of production quality is an example of such a metaclass.

```
class DocMeta(type):

    def __init__(self, name, bases, attrs):
        for key, value in attrs.items():
            # skip special and private methods
            if key.startswith("__"):
                continue
            # skip any non-callable
            if not hasattr(value, "__call__"):
                continue
            # check for a doc string. a better way may be to store
            # all methods without a docstring then throw an error showing
            # all of them rather than stopping on first encounter
            if not getattr(value, '__doc__'):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self, name, bases, attrs)
```

`DocMeta` is a `type` subclass that overrides the `type` class `__init__` method. The implemented `__init__` method iterates through all the class attributes searching for non-private methods missing a *docstring*; if such is encountered an exception is thrown as shown below.

```
class Car(object, metaclass=DocMeta):

    def __init__(self, make, model, color):
        self.make = make
```

```
            self.model = model
            self.color = color

        def change_gear(self):
            print("Changing gear")

        def start_engine(self):
            print("Changing engine")

car = Car()
Traceback (most recent call last):
  File "abc.py", line 47, in <module>
    class Car(object):
  File "abc.py", line 42, in __init__
    raise TypeError("%s must have a docstring" % key)
TypeError: change_gear must have a docstring
```

Another trivial example that illustrates an application of a metaclass is in the creation of a final class, that is a class that cannot be sub-classed. Some people may argue that final classes are *unpythonic* but for illustration purposes such functionality is implemented using a metaclass in the following snippet.

```
class Final(type):

    def __init__(cls, name, bases, namespace):
        super().__init__(name, bases, namespace)
        for c in bases:
            if isinstance(c, Final):
                raise TypeError(c.__name__ + " is final")

class B(object, metaclass=Final):
    pass

class C(B):
    pass

>>> class B(object, metaclass=Final):
...     pass
...
>>> class C(B):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __init__
TypeError: B is final
```

In the example, the metaclass simply performs a check ensuring that the final class is never part of the base classes for any class being created.

Another very good example of a metaclass in action is in *Abstract Base Classes* that was previously discussed. When defining an abstract base class, the `ABCMeta` metaclass from the `abc` module is used as the metaclass for the abstract base class being defined and the `@abstractmethod` and `@abstractproperty` decorators are used to create methods and properties that must be implemented by non-abstract subclasses.

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
     __metaclass__ = ABCMeta

     @abstractmethod
     def change_gear(self):
          pass

     @abstractmethod
     def start_engine(self):
          pass


class Car(Vehicle):

     def __init__(self, make, model, color):
          self.make = make
          self.model = model
          self.color = color

# abstract methods not implemented
>>> car = Car("Toyota", "Avensis", "silver")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Car with abstract methods change_gear, start_engine
>>>
```

Once a class implements all abstract methods then such a class becomes a concrete class and can be instantiated by a user.

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
     __metaclass__ = ABCMeta

     @abstractmethod
     def change_gear(self):
          pass

     @abstractmethod
     def start_engine(self):
          pass


class Car(Vehicle):

     def __init__(self, make, model, color):
          self.make = make
          self.model = model
          self.color = color

     def change_gear(self):
          print("Changing gear")

     def start_engine(self):
          print("Changing engine")
```

```
>>> car = Car("Toyota", "Avensis", "silver")
>>> print(isinstance(car, Vehicle))
True
```

Overriding `__new__` vs `__init__` in Custom Metaclasses

Sometimes, there is confusion over whether to override the `__init__` or `__new__` method when defining custom metaclasses. The decision about which to override depends the objective of such custom metaclasses. If the intent is to modify the class by changing some class attribute such as the list of base classes then the `__new__` method should be overridden. The following example is a metaclass that checks for *camel case* attribute names and converts such to names with underscores between words.

```
class NamingMeta(type):

    def __new__(mcl, name, bases, attrs):
        new_attrs = dict()
        for key, value in attrs.items():
            updated_name = NamingMeta.convert(key)
            new_attrs[updated_name] = value
        return super(NamingMeta, mcl).__new__(mcl, name, bases, new_attrs)

    @staticmethod
    def convert(name):
        s1 = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', name)
        return re.sub('([a-z0-9])([A-Z])', r'\1_\2', s1).lower()
```

It would not be possible to modify class attributes such as the list of base classes or attribute names in the `__init__` method because as has been said previously, this method is called after the object has already been created. On the other hand, when the intent is just to carry out initialization or validation checks such as was done with the `DocMeta` and `Final` metaclasses then the `__init__` method of the metaclass should be overridden.

8.4 Context Managers

Sometimes, there is a need to execute some operations between another *pair of operations*. For example, open a file, **read from the file** and close the file or acquire a lock on a data structure, **work with the data structure** and release the data structure. These kinds of requirements come up most especially when dealing with system resources where the resource is acquired, worked with and then released. It is important that the acquisition and release of such resources are handled carefully so that any errors that may occur are correctly handled. Writing code to handle this all the time leads to a lot of repetition and cumbersome code. ***Context managers*** provide a solution to this. They provide a mean for abstracting away a pair of operations that are executed before and after another group of operation using the `with` statement. The `with` statement enables a set of operations to run within a context. The context is controlled by a context manager object. An example of the use of the `with` statement is in opening files; this involves a pair of operations - opening and closing the file.

```
# create a context
with open('output.txt', 'w') as f:
```

```
    # carry out operations within context
    f.write('Hi there!')
```

The `with` statement can be used with any object that implements the *context management protocol*. This protocol defines a set of operations, `__enter__` and `__exit__` that are executed just before the start of execution of some piece of code and after the end of execution of some piece of code respectively. Generally, the definition and use of a context manager is shown in the following snippet.

```
class context:
     def __enter__(self):
          set resource up
          return resource

     def __exit__(self, type, value, traceback):
          tear resource down

# the context object returned by __enter__ method is bound to name
with context() as name:
      do some functionality
```

If the initialised resource is used within the context then the `__enter__` method must return the resource object so that it is bound within the `with` statement using the `as` mechanism. A resource object must not be returned if the code being executed in the context doesn't require a reference to the object that is set-up. The following is a very trivial example of a class that implements the *context management protocol* in a very simple fashion.

```
>>> class Timer:
...       def __init__(self):
...            pass
...       def __enter__(self):
...            self.start_time = time.time()
...       def __exit__(self, type, value, traceback):
...            print("Operation took {} seconds to complete".format(time.time()-self.start_time))
...


...
>>> with Foo():
...       print("Hey testing context managers")
...
Hey testing context managers
Operation took 0.00010395050048828125 seconds to complete
>>>
```

When the `with` statement executes, the `__enter__()` method is called to create a new context; if a resource is initialized for use here then it is returned but this is not the case in this example. After the operations within the context are executed, the `__exit__()` method is called with the `type`, `value` and `traceback` as arguments. If no exception is raised during the execution of the of the operations within the context then all arguments are set to `None`. The `__exit__` method returns a `True` or `False` depending on whether any raised exceptions have been handled. When `False` is returned then exception raised are propagated outside of the context for other code blocks to handle. Any resource clean-up

is also carried out within the `__exit__()` method. This is all there is to context management. Now rather than write `try…finally` code to ensure that a file is closed or that a lock is released every time such resource is used, such chores can be handled in the the `__exit__` method of a context manager class thus eliminating code duplication and making the code more intelligible.

The `Contextlib` module

For very simple use cases, there is no need to go through the hassle of implementing our own classes with `__enter__` and `__exit__` methods. The python `contextlib` module provides us with a high level method for implementing context manager. To define a context manager, the `@contextmanager` decorator from the `contextlib` module is used to decorate a function that handles the resource in question or carries out any initialization and clean-up; this function carrying out the initialization and tear down must however be a generator function. The following example illustrates this.

```
from contextlib import contextmanager

>>> from contextlib import contextmanager
>>> @contextmanager
... def time_func():
...     start_time = time.time()
...     yield
...     print("Operation took {} seconds".format(time.time()-start_time))


>>> with time_func():
...     print("Hey testing the context manager")
...
Hey testing the context manager
Operation took 7.009506225585938e-05 seconds
```

This context generator function, `time_func` in this case, must yield exactly one value if it is required that a value be bound to a name in the `with` statement's `as` clause. When generator yields, the code block nested in the `with` statement is executed. The generator is then resumed after the code block finishes execution. If an exception occurs during the execution of a block and is not handled in the block, the exception is re-raised inside the generator at the point where the `yield` occurred. If an exception is caught for purposes other than adequately handling such an exception then the generator must re-raise that exception otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume normally after the context block.

Context managers just like decorators and metaclasses provide a clean method for abstracting away these kind of repetitive code that can clutter code and makes following code logic difficult.

9. Modules And Packages

Modules and packages are the last organizational unit of code that are discussed. They provide the means by which large programs can be developed and shared.

9.1 Modules

Modules enable the reuse of programs. A *module* is a file that contains a collection of definitions and statements and has a `.py` extension. The contents of a module can be used by importing the module either into another module or into the interpreter. To illustrate this, our favourite `Account` class shown in the following snippet is saved in a module called `account.py`.

```
class Account:
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def deposit(self, amt):
          self.balance = self.balance + amt

     def withdraw(self, amt):
          self.balance = self.balance - amt

     def inquiry(self):
          return self.balance
```

To re-use the module definitions, the `import` statement is used to import the module as shown in the following snippet.

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import account
>>> acct = account.Account("obi", 10)
>>> acct
<account.Account object at 0x101b6e358>
>>>
```

All executable statements contained within a module are executed when the module is imported. A module is also an object that has a type - `module` as such all generic operations that apply to objects can be applied to modules. The following snippets show some unintuitive ways of manipulating module objects.

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import account
>>> type(account)
<class 'module'>
>>> getattr(account, 'Account') # access the Account class using getattr
<class 'cl.Account'>
>>> account.__dict__
{'json': <module 'json' from '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/j
on/__init__.py'>, '__cached__': '/Users/c4obi/writings/scratch/src/__pycache__/cl.cpython-34.pyc', '
__loader__': <_frozen_importlib.SourceFileLoader object at 0x10133d4e0>, '__doc__': None, '__file__
: '/Users/c4obi/writings/scratch/src/cl.py', 'Account': <class 'account.Account'>, '__package__': ''
, '__builtins__': { ...} ...
      }
```

Each module possesses its own unique global namespace that is used by all functions and classes defined within the module and when this feature is properly used, it eliminates worries about name clashes from third party modules. The `dir()` function without any argument can be used within a module to find out what names are available in a module's namespace.

As mentioned, a module can import another module; when this happens and depending on the form of the `import`, the imported module's name, part of the name defined within the imported module or all names defined with the imported module could be placed in the namespace of the module doing the importing. For example, `from account import Account` imports and place the `Account` name from the `account` module into the namespace, `import account` imports and adds the `account` name referencing the whole module to the namespace while `from account import *` will import and add all names in the account module except those that start with an underscore to the current namespace. Using `from module import *` as a form of import is strongly advised against as it may import names that the developer is not aware of and that conflict with names used in the module doing the importing. Python has the `__all__` special variable that can be used within modules. This value of the `__all__` variable should be a list that contains the names within a module that are imported from such module when the `from module import *` syntax is used. Defining this method is totally optional on the part of the developer. We illustrate the use of the `__all__` special method with the following example.

```
__all__ = ['Account']

class Account:
     num_accounts = 0

     def __init__(self, name, balance):
          self.name = name
          self.balance = balance
          Account.num_accounts += 1

     def del_account(self):
          Account.num_accounts -= 1

     def deposit(self, amt):
          self.balance = self.balance + amt
```

```
        def withdraw(self, amt):
            self.balance = self.balance - amt

        def inquiry(self):
            return self.balance

    class SharedAccount:
        pass

    >>> from account import *
    >>> dir()
    ['Account', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__'] # o
nly Account has been imported
    >>>
```

The name of an imported module is gotten by referencing the `__name__` attribute of the imported module. In the case of a module that is currently executing, the `__name__` value is set to `__main__`. Python modules can be executed with `python module <arguments>`. A corollary of the fact that the `__name__` of the currently executing module is set to `__main__` is that we can have a recipe such as the following.

```
    if __name__ == "__main__":
        # run some code
```

That makes the module usable as a standalone script as well as an importable module. A popular use of the above recipe is for running unittest; we can run the module as a standalone to test it but then import it for use into another module without running the test cases.

Reloading Modules

Once modules have been imported into the interpreter, any change to such a module is not reflected within the interpreters. However, Python provides the `importlib.reload` that can be used to re-import a module once again into the current namepace.

9.2 How are Modules found?

Import statements are able to import modules that are in any of the paths given by the `sys.path` variable. The import system uses a greedy strategy in which the first module found is imported. The content of the `sys.path` variable is unique to each Python installation. An example of the value of the `sys.path` variable on a *Mac* operating system is shown in the following snippet.

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python34.zip', '/Library/Frameworks/Pyth\
on.framework/Versions/3.4/lib/python3.4', '/Library/Frameworks/Python.framework/Versions/3.4/lib/pyt\
hon3.4/plat-darwin', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/lib-dynload', \
'/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages']
```

The `sys.path` list can be modified at runtime by adding or removing elements from this list. However, when the interpreter is started conventionally, the `sys.path` list contains

paths that come from three sources namely: `sys.prefix`, *PYTHONPATH* and initialization by the `site.py` module.

1. `sys.prefix`: This variable specifies the base location for a given Python installation. From this base location, the Python interpreter can work out the location of the Python standard library modules. The location of the standard library is given by the following paths.

```
sys.prefix + '/lib/python3X.zip'
sys.prefix + '/lib/python3.X'
sys.prefix + '/lib/python3.X/plat-sysname'
sys.exec_prefix + '/lib/python3.X/lib-dynload'
```

The paths of the standard library can be found by running the Python interpreter with the `-S` option; this prevents the `site.py` initialization that adds the third party package paths to the `sys.path` list. The location of the standard library can also be overriden by defining the *PYTHONHOME* environment variable that replaces the `sys.prefix` and `sys.exec_prefix`.

1. *PYTHONPATH*: Users can define the *PYTHONPATH* environment variable and the value of this variable is added as the first argument to the `sys.path` list. This variable can be set to the directory where a user keeps user defined modules.
2. `site.py`: This is a path configuration module that is loaded during the initialization of the interpreter. This modules adds site-specific paths to the module search path. The `site.py` starts by constructing up to four directories from a prefix and a suffix. For the prefix, it uses `sys.prefix` and `sys.exec_prefix`. For the suffix, it uses the empty string and then `lib/site-packages` on Windows or `lib/pythonX.Y/site-packages` on Unix and Macintosh. For each of these distinct combinations, if it refers to an existing directory, it is added to the `sys.path` and further inspected for configuration files. The configuration files are files with `.pth` extension. The contents are additional items one per line to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. Each item is added to `sys.path`once. Blank lines and lines beginning with # are skipped. Lines starting with `import` followed by space or tab are executed. After these path manipulations, an attempt is made to import a module named `sitecustomize` that can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an `ImportError` exception, it is silently ignored. After this, if `ENABLE_USER_SITE` is true, an attempt is made to import a module named `usercustomize` that can perform arbitrary user-specific customizations, . This file is intended to be created in the user site-packages directory that is part of `sys.path` unless disabled by `-s`. Any `ImportError` is silently ignored.

9.3 Packages

Just as modules provide a mean for organizing statements and definitions, *packages* provide a mean for organizing modules. A close but imperfect analogy of the relationship of packages to modules is that of folders to files on computer file systems. A package just

like a folder can be composed of a number of module files. In Python however, packages are just like modules; in fact all packages are modules but not all modules are packages. The difference between a module and package is the presence of a `__path__` special variable in a package object that does not have a `None` value. Packages can have sub-packages and so on; when referencing a package and it corresponding sub-packages the dot notation is used so a *complex* number sub-package within a *mathematics* package will be referenced as `math.complex`.

There are currently two types of packages:- regular packages and namespace packages.

Regular Packages

A regular package is one that consists of a group of modules in a folder with an `__init__.py` module within the folder. The presence of this `__init__.py` file within the folder cause the interpreter to treat the folder as a package. An example of package structure is the following.

```
parent/          <----- folder
     __init__.py
     one/          <------ sub-folder
          __init__.py
          a.py
     two/          <------ sub-folder
          __init__.py
          b.py
```

The *parent*, *one* and *two* folders are all packages because they all contain an `__init__.py` module within each of their respective folders. `one` and `two` are sub-packages of the `parent` package. Whenever a package is imported, the `__init__.py` module of such a package is executed. One can think of the `__init__.py` as the store of attributes for the package - only symbols defined in this module are attributes of the imported module. Assuming the `__init__.py` module from the above `parent` package is empty and the package, `parent`, is imported using `import parent`, the `parent` package will have no module or subpackage as an attribute. The following code listing shows this.

```
>>> import parent
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'parent']
>>> dir(parent)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package
', '__path__', '__spec__']
```

As the example shows, none of the modules or sub-packages is listed as an attribute of the imported package object. On the other hand, if a symbol, `package="testing packages"`, is defined in the `__init__.py` module of the parent package and the parent package is imported, the package object has this symbol as an attribute as shown in the following code listing .

```
>>> import parent
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'parent']
>>> dir(parent)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package
```

```
', '__path__', '__spec__', 'package']
          >>> parent.package
          'testing packages'
          >>>
```

When a sub-package is imported, all `__init__.py` modules in parent packages are imported in addition to the `__init__.py` module of the sub-package. Sub-packages are referenced during import using the dot notation just like modules in packages are. In the previous package structure, the notation would be `parent.one` to reference the `one` sub-package. Packages support the same kind of `import` semantics as modules; individual modules or packages can be imported as in the following example.

```
# import the module a
import parent.one.a
```

When the above method is used then the fully qualified name for the module, `parent.one.a`, must be used to access any symbol in the module. Note that when using this method of import, the last symbol can be either a module or sub-package only; classes, functions or variables defined within modules are not allowed. It is also possible to `import` just the module or sub-package that is needed as the following example shows.

```
# importing just required module
from parent.one import a

# importing just required sub-package
from parent import one
```

Symbols defined in the `a` module or modules in the `one` package can then be referenced using dot notation with just `a` or `one` as the prefix. The import forms, `from package import *` or `from package.subpackage import *`, can be used to import all the modules in a package or sub-package. This form of import should however be used carefully if ever used as it may import some names into the namespace that may cause naming conflicts. Packages support the `__all__` (the value of this should by convention be a list) variable for listing modules or names that are visible when the package is imported using the `from package import *` syntax. If `__all__` is not defined, the statement `from package import *` does not import all submodules from the package into the current namespace rather it only ensures that the package has been imported possibly running any initialization code in `__init__.py` and then imports whatever symbols are defined in the `__init__.py` module; including any names defined here and submodules imported here.

Namespace Packages

A namespace package is a package in which the component modules and sub-packages of the package may reside in multiple different locations. The various components may reside on different part of the file system, in zip files, on the network or on any other location searched by interpreter during the import process however when the package is imported, all components exist in a common namespace. To illustrate a namespace package, observe the following directory structures containing modules; both directories, `apollo` and `gemini` could be located on any part of the file system and not necessarily next to each other.

```
apollo/
     space/
          test.py
gemini/
     space/
          test1.py
```

In these directories, the name, `space`, is used as a common namespace and will serve as the package name. Observe the absence of `__init__.py` modules in either directory. The absence of this module within these directories is a signal to the interpreter that it should create a namespace package when it encounters such. To be able to import this `space` package, the paths for its components must first of all be added to interpreter's module search path, `sys.path`.

```
>>> import sys
>>> sys.path.extend(['apollo', 'gemini'])
>>> import space.test
>>> import space.test1
```

Observe that the two different package directories are now logically regarded as a single name space and either `space.test` or `space.test1` can be imported as if they existed in the same package. The key to a namespace package is the absence of the `__init__.py` modules in the top-level directory that serves as the common namespace. The absence of the `__init__.py` module causes the interpreter to create a list of all directories within its `sys.path` variable that contain a matching directory name rather than throw an exception. A special namespace package module is then created and a read-only copy of the list of directories is stored in its `__path__` variable. The following code listing gives an example of this.

```
>>> space.__path__
_NamespacePath(['apollo/space', 'gemini/space'])
```

Namespaces bring added flexibility to package manipulation because namespaces can be extend by anyone with their own code thus eliminating the need to modify package structures in third party packages. For example, suppose a user had his or her own directory of code like this:

```
my-package/
     space/
          custom.py
```

Once this directory is added to `sys.path` along with the other packages, it would seamlessly merge together with the other `space` package directories and the contents can also be imported along with any existing artefacts.

```
>>> import space.custom
>>> import space.test
>>> import space.test1
```

9.4 The Import System

The `import` statement and `importlib.import_module()` function provide the required *import* functionality in Python. A call to the `import` statement combines two actions:

1. A search operation to find the requested module through a call to the `__import__` statement and
2. A binding operation to add the module returned from operation 1 to the current namespace.

If the `__import__` call does not find the requested module then an `ImportError` is returned.

The Import Search Process

The import mechanism uses the fully qualified name of the module for the search. In the case that the fully qualified name is a sequence of names separated by dots e.g `foo.bar.baz`, the interpreter will attempt to import `foo` followed by `bar` followed by `bar`. If any of these modules is not found then an `ImportError` is raised.

The `sys.modules` variable is a cache for all previously imported modules and is the first port of call in the module search process. If a module is present in the `sys.modules` cache then it is returned otherwise an `ImportError` is raised and the search continues. The `sys.modules` cache is writeable so user code can manipulate the content of the cache. An example of the content of the cache is shown in the following snippet.

```
>>> import sys
>>> sys.modules
{'readline': <module 'readline' from '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.
4/lib-dynload/readline.so'>, 'json.scanner': <module 'json.scanner' from '/Library/Frameworks/Python
.framework/Versions/3.4/lib/python3.4/json/scanner.py'>, '_sre': <module '_sre' (built-in)>, 'copyre
g': <module 'copyreg' from '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/copyreg.
py'>, '_collections_abc': <module '_collections_abc' from '/Library/Frameworks/Python.framework/Vers
ions/3.4/lib/python3.4/_collections_abc.py'>, 'cl': <module 'cl' from '/Users/c4obi/writings/scratch
/src/cl.py'>, 'rlcompleter': <module 'rlcompleter' from '/Library/Frameworks/Python.framework/Versio
ns/3.4/lib/python3.4/rlcompleter.py'>, '_sitebuiltins': <module '_sitebuiltins' from '/Library/Frame
works/Python.framework/Versions/3.4/lib/python3.4/_sitebuiltins.py'>, '_imp': <module '_imp' (built-
in)>, '_json': <module '_json' from '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4
/lib-dynload/_json.so'>, '_weakrefset': <module '_weakrefset' from '/Library/Frameworks/Python.frame
work/Versions/3.4/lib/python3.4/_weakrefset.py'>, 'json.decoder': <module 'json.decoder' from '/Libr
ary/Frameworks/Python.framework/Versions/3.4/lib/python3.4/json/decoder.py'>, '_codecs': <module '_c
odecs' (built-in)>, 'codecs': <module 'codecs' from '/Library/Frameworks/Python.framework/Versions/3
.4/lib/python3.4/codecs.py'>, ... }
```

Finders and Loaders

When a module is not found in the cache, the interpreter makes use of its *import protocol* to try and find the module. The Python *import protocol* defines ***finder*** and ***loader*** objects. Objects that implement both interfaces are called ***importers***.

Finders define strategies for locating modules. Modules maybe available locally on the file system in regular files or in zipped files, or in other locations such as a database or even at a remote location. Finders have to be able to deal with such locations if modules are going to be imported from any of such locations. By default, Python has support for finders that handle the following scenarios.

1. Built-in modules,
2. Frozen modules and
3. Path based modules - this finder handles imports that have to interact with the import path given by the `sys.path` variable as shown in the following.

These finders are located in the `sys.meta_path` variable as shown in the following snippet.

```
>>> import sys
>>> sys.meta_path
[<class '_frozen_importlib.BuiltinImporter'>, <class '_frozen_importlib.FrozenImporter'>, <class '_f\
rozen_importlib.PathFinder'>]
>>>
```

The interpreter continues the search for the module by querying each finder in the `meta_path` to find which can handle the module. The finder objects must implement the `find_spec` method that takes three arguments: the first is the fully qualified name of the module, the second is an import path that is used for the module search - this is `None` for top level modules but for sub-modules or sub-packages, it is the value of the parent package's `__path__` and the third argument is an existing module object that is passed in by the system only when a module is being reloaded.

If one of the *finders* locates the module, it returns a *module spec* that is used by the interpreter import machinery to create and load the module (loading is tantamount to executing the module). The *loaders* carry out the module execution in the module's global namespace. This is done by a call to the `importlib.abc.Loader.exec_module()` method with the already created module object as argument.

Customizing the import process

The import process can be customized via import hooks. There are two types of this hook: *meta hooks* and *import path hooks*.

Meta hooks

These are called at the start of the import process immediately after the `sys.modules` cache lookup and before any other process. These hooks can override any of the default finders search processes. Meta hooks are registered by adding new *finder* objects to the `sys.meta_path` variable.

To understand how a custom `meta_path` hook can be implemented, a very simple case is illustrated. In online Python interpreters, some built-in modules such as the `os` are disabled or restricted to prevent malicious use. A very simple way to implement this is to implement a meta import hook that raises an exception any time a restricted import is attempted; the following snippet shows such an example.

```
class RestrictedImportFinder:

    def __init__(self):
        self.restr_module_names = ['os']

    def find_spec(self, fqn, path=None, module=None):
        if fqn in self.restr_module_names:
            raise ImportError("%s is a restricted module and cannot be imported" % fqn)
```

```
            return None


    import sys
    # remove os from sys.module cache
    del sys.modules['os']
    sys.meta_path.insert(0, RestrictedImportFinder())
    import os

    Traceback (most recent call last):
      File "test_concat.py", line 16, in <module>
        import os
      File "test_concat.py", line 9, in find_spec
        raise ImportError("%s is a restricted module and cannot be imported" % fqn)
    ImportError: os is a restricted module and cannot be imported
```

Import Path hooks

These hooks are called as part of the `sys.path` or `package.__path__` processing. Recall from our previous discussion that a path based finder is one of the default meta-finder and this finder works with entries in the `sys.path` variable. The meta path based finder delegates the job of finding modules on the `sys.path` variables to other finders - these are the *import path hooks*. The `sys.path_hooks` is a collection of built in path entry finders. By default, the Python interpreter has support for processing files in zip folders and normal files in directories as shown in the following snippet.

```
import sys
>>> sys.path_hooks
[<class 'zipimport.zipimporter'>, <function FileFinder.path_hook.<locals>.path_hook_for_FileFinder a\
t 0x1003c1b70>]
```

Each hooks knows how to handle a particular kind of file. For example, an attempt to get the finder for one of the entries in `sys.path` is attempted in the following snippet.

```
    >>> sys.path_hooks
    [<class 'zipimport.zipimporter'>, <function FileFinder.path_hook.<locals>.path_hook_for_FileFind
er at 0x1003c1b70>]
    # sys.prefix is a directory
    >>> path = sys.prefix
    # sys.path_hooks[0] is associated with zip files
    >>> finder = sys.path_hooks[0](path)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    zipimport.ZipImportError: not a Zip file
    >>> finder = sys.path_hooks[1](path)
    >>> finder
    FileFinder('/Library/Frameworks/Python.framework/Versions/3.4')
    >>>
```

New *import path* hooks can be added by inserting new callables into the `sys.path_hooks`.

Why You Probably Should Not Reload Modules…

Now that we understand that the last step of a module import is the `exec` of the module code within the global namespace of the importing module, it is clearer why it maybe a bad idea to use the `importlib.reaload` to reload modules that have changed.

A module reload does not purge the global namespace of objects from the module being imported. Imagine a module, `Foo`, that has a function, `print_name` imported into another module, `Bar`; the function, `Foo.print_name`, is referenced by a variable, `x`, in the module, `Bar`. Now if the implementation for `print_name` is changed for some reason and then reloaded in `Bar`, something interesting happens. Since the reload of the module `Foo` will cause an `exec` of the module contents without any prior clean-up, the reference that `x` holds to the previous implementation of `Foo.print_name` will persist thus we have *two* implementations and this is most probably not the behaviour expected.

For this reason, reloading a module is something that maybe worth avoiding in any sufficiently complex Python program.

9.5 Distributing Python Programs

Python provides the `distutils` module for packaging up Python code for distribution. Assuming the program has been properly written, documented and structured then distributing it is relatively straightforward using `distutils`. One just has to:

```
1. write a setup script (setup.py by convention)
2. (optional) write a setup configuration file
3. create a source distribution
4. (optional) create one or more built (binary) distributions
```

A set-up script using `distutils` is a `setup.py` file. For a program with the following package structure,

```python
parent/
     __init__.py
     spam.py
     one/
          __init__.py
          a.py
     two/
          __init__.py
          b.py
```

an example of a simple `setup.py` file is given in the following snippet.

```
from distutils.core import setup
setup(name='parent',
        version='1.0',
        author="xxxxxx",
        maintainer="xxxx",
        maintainer_email="xxxxx"
        py_modules=['spam'],
        packages=['one', 'two'],
        scripts=[]
        )
```

The `setup.py` file must exist at the top level directory so in this case, it should exist at `parent/setup.py`. The values used in the set-up script are self explanatory. `py_modules` will contain the names of all single file python modules, packages will contains a list of all

packages,scripts will contain a list of all scripts within the program. The rest of the arguments though not exhaustive of the possible parameters are self explanatory.

Once the `setup.py` file is ready, the following snippet is used at the commandline to create an archive file for distribution.

```
>>> python setup.py sdist
```

`sdist` will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script `setup.py`, your modules and packages. The archive file will be named parent-1.0.tar.gz (or .zip), and will unpack into a directory `parent-1.0.`. To install the created distribution, the file is unzipped and `python setup.py install` is run inside the directory. This will install the package in the `site-packages` directory for the installation.

One can also create one or more built distributions for programs. For instance, if running a Windows machine, one can make the use of the program easy for end users by creating an executable installer with the `bdist_wininst` command. For example:

```
python setup.py bdist_wininst
```

will create an executable installer, `parent-1.0.win32.exe`, in the current directory.

Other useful built distribution formats are RPM, implemented by the `bdist_rpm`command, `bdist_pkgtool` for Solaris, and `bdist_sdux` for HP-UX install. It is important to note that the use of `distutils` assumes that the end user of a distributed package will have the python interpreter already installed.

10. Inspecting Objects

The Python `inspect` module provides powerful methods for interacting with live Python objects. The methods provided by this module help with *type checking*, *sourcecode retrieval*, *class and function inspection* and *Interpreter stack inspection*. The documentation is the golden source of informtion for this module but a few of the classes and methods in this module are discussed to show the power of this module.

10.1 Handling source code

The `inspect` module provides functions for accessing the source code of functions, classes and modules. All examples are carried out using our `Account` class as defined in the following snippet.

```
# python class for intermediate python book

class Account:
"""base class for representing user accounts"""
num_accounts = 0

def __init__(self, name, balance):
     self.name = name
     self.balance = balance
     Account.num_accounts += 1

def del_account(self):
     Account.num_accounts -= 1

def __getattr__(self, name):
     """handle attribute reference for non-existent attribute"""
     return "Hey I dont see any attribute called {}".format(name)

def deposit(self, amt):
     self.balance = self.balance + amt

def withdraw(self, amt):
     self.balance = self.balance - amt

def inquiry(self):
     return "Name={}, balance={}".format(self.name, self.balance) =
```

Some of the methods from the `inspect` module for handling source code include:

1. `inspect.getdoc(object)`: This returns the documentation string for the argument object. The string returned is cleaned up with `inspect.cleandoc()`.

```
>>> acct = test.Account("obi", 1000000)
>>> import inspect
>>> inspect.getdoc(acct)
```

```
'base class for representing user accounts'
>>>
```

2. `inspect.getcomments(object)`: This returns a single string containing any lines of comments. For a class, function or method these are comments immediately preceding the argument object's source code while for a module, it is the comments at the top of the Python source file.

```
>>> import test
>>> import inspect
>>> acct = test.Account("obi", 1000000)
>>> inspect.getcomments(acct)
>>> inspect.getcomments(test)
'# python class for intermediate python book\n'
```

3. `inspect.getfile(object)`: Return the name of the file in which an object was defined. The argument should be a module, class, method, function, traceback, frame or code object. This will fail with a `TypeError` if the object is a built-in module, class, or function.

```
>>> inspect.getfile(test.Account)
'/Users/c4obi/src/test_concat.py'
>>>
```

4. `inspect.getmodule(object)`: This function attempts to guess the module that the argument object was defined in.

```
>>> inspect.getmodule(acct)
<module 'test' from '/Users/c4obi/src/test.py'>
>>>
```

5. `inspect.getsourcefile(object)`: This returns the name of the Python source file in which the argument object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

```
>>> inspect.getsourcefile(test_concat.Account)
'/Users/c4obi/src/test.py'
>>>
```

6. `inspect.getsourcelines(object)`: This returns a tuple of the list of source code lines and the line number on which the source code for the argument object begins. The argument may be a module, class, method, function, traceback, frame, or code object. An `OSError` is raised if the source code cannot be retrieved.

```
>>> inspect.getsourcelines(test_concat.Account)
(['class Account:\n', '    """base class for representing user accounts"""\n', '    num_accoun
0\n', '\n', '    def __init__(self, name, balance):\n', '        self.name = name \n', '
.balance = balance \n', '        Account.num_accounts += 1\n', '\n', '    def del_account(self)
', '        Account.num_accounts -= 1\n', '\n', '    def __getattr__(self, name):\n', '        "'
le attribute reference for non-existent attribute"""\n', '        return "Hey I dont see any at
te called {}".format(name)\n', '\n', '    def deposit(self, amt):\n', '        self.balance = s
alance + amt \n', '\n', '    def withdraw(self, amt):\n', '        self.balance = self.balance
 \n', '\n', '    def inquiry(self):\n', '        return "Name={}, balance={}".format(self.name,
.balance) \n'], 52)
```

7. `inspect.getsource(object)`: Return the human readable text of the source code for the argument object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `OSError` is raised if the source code cannot be retrieved. Note the difference between this and `inspect.getsourcelines` is that this method returns the source code as a single string while `inspect.getsourcelines` returns a list of source code lines.

```
 >>> inspect.getsource(test.Account)
 'class Account:\n    """base class for representing user accounts"""\n    num_accounts = 0\n\n
ef __init__(self, name, balance):\n        self.name = name \n        self.balance = balance \n
   Account.num_accounts += 1\n\n    def del_account(self):\n        Account.num_accounts -= 1\n\
 def __getattr__(self, name):\n        """handle attribute reference for non-existent attribute'
        return "Hey I dont see any attribute called {}".format(name)\n\n    def deposit(self, am
n          self.balance = self.balance + amt \n\n     def withdraw(self, amt):\n          sel
self.balance - amt \n\n     def inquiry(self):\n          return "Name={}, balance={}".format(
, self.balance) \n'
 >>>
```

8. `inspect.cleandoc(doc)`: This cleans up indentation from documentation strings that have been indented to line up with blocks of code. Any white-space that can be uniformly removed from the second line onwards is removed and all tabs are expanded to spaces.

10.2 Inspecting Classes and Functions

The `inspect` module provides some classes and functions for interacting with classes and functions. `Signature`, `Parameter` and `BoundArguments` are important classes in the the inspect module.

1. `Signature`: This can be used to represent the call signature and return annotation of a function or method. A `Signature` object can be obtained by calling the `inspect.signature` method with a function or method as argument. Each parameter accepted by the function or method is represented as a `Parameter` object in the parameter collection of the `Signature` object. `Signature` objects support the `bind` method for mapping from positional and keyword arguments to parameters. The `bind(*args, **kwargs)` method will return a `BoundsArguments` object if `*args` and `**kwargs` match the `signature` else it raises a `TypeError`. The `Signature` class also has the `bind_partial(*args, **kwargs)` method that works in the same way as `Signature.bind` but allows the omission of some arguments.

```
 >>> def test(a, b:int) -> int:
 ...       return a^2+b
 ...
 >>> inspect.signature(test)
 <inspect.Signature object at 0x101b3c518>
 >>> sig = inspect.signature(test)
 >>> dir(sig)
 ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getat
e__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
duce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__
sshook__', '_bind', '_bound_arguments_cls', '_parameter_cls', '_parameters', '_return_annotati
bind', 'bind_partial', 'empty', 'from_builtin', 'from_function', 'parameters', 'replace', 'ret
```

```
notation']
 >>> sig.parameters
 mappingproxy(OrderedDict([('a', <Parameter at 0x101cbf708 'a'>), ('b', <Parameter at 0x101cbf
'>)]))
 >>> str(sig)
 '(a, b:int) -> int'
```

2. `Parameter`: `Parameter` objects represent function or method arguments within a `Signature`. Using the previous example for illustration, the parameters of a signature can be accessed as shown in the following snippet.

```
 >>> sig.parameters['a']
 <Parameter at 0x101cbf708 'a'>
 >>> sig.parameters['b']
 <Parameter at 0x101cbf828 'b'>
 >>> type(sig.parameters['a']
 ... )
 <class 'inspect.Parameter'>
 >>> dir(sig.parameters['a'])
 ['KEYWORD_ONLY', 'POSITIONAL_ONLY', 'POSITIONAL_OR_KEYWORD', 'VAR_KEYWORD', 'VAR_POSITIONAL',
ass__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__rec
, '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclas
_', '_annotation', '_default', '_kind', '_name', 'annotation', 'default', 'empty', 'kind', 'na
replace']
```

Important attributes of a `Parameter` object are the `name` and `kind` attributes. The kind attribute is a string that could either be `POSITIONAL_ONLY`, `POSITIONAL_OR_KEYWORD`, `VAR_POSITIONAL`, `KEYWORD_ONLY` or `VAR_KEYWORD`.

3. `BoundArguments`: This is the return value of a `Signature.bind` or `Signature.partial_bind` method call.

```
>>> sig
<inspect.Signature object at 0x101b3c5c0>
>>> sig.bind(1, 2)
<inspect.BoundArguments object at 0x1019e6048>
```

A `BoundArguments` objects contains the mapping of arguments to function or method parameters.

```
 >>> arg = sig.bind(1, 2)
 >>> dir(arg)
 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge_
_getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__'
ew__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__s
shook__', '__weakref__', '_signature', 'args', 'arguments', 'kwargs', 'signature']
 >>> arg.args
 (1, 2)
 >>> arg.arguments
 OrderedDict([('a', 1), ('b', 2)])
```

A `BoundArguments` object has the following attributes.

1. `args`: this is a tuple of postional parameter argument values.

2. `arguments`: this is an ordered mapping of parameter argument names to parameter argument values.
3. `kwargs`: this is a `dict` of keyword argument values.
4. `signature`: this is a reference to the parent `Signature` object.

Interesting functionality can be implemented by making use of these classes mentioned above. For example, we can implement a rudimentary `type checker` decorator for a function by making use of these classes as shown in the following snippet (thanks to the python cookbook for this).

```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
     def decorate(func):
          # If in optimized mode, disable type checking
          if not __debug__:
               return func
               # Map function argument names to supplied types
               sig = signature(func)
               bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

          @wraps(func)
          def wrapper(*args, **kwargs):
               bound_values = sig.bind(*args, **kwargs)
               # Enforce type assertions across supplied arguments for name, value in bound_valu
arguments.items():
               if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                         raise TypeError('Argument {} must be {}'.format(name,bound_types[name])
               return func(*args, **kwargs)
          return wrapper
     return decorate
```

The defined decorator, `type_assert`, can then be used to enforce type assertion as shown in the following example.

```
>>> @typeassert(str)
... def print_name(name):
...      print("My name is {}".format(name))
...
>>> print_name(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/c4obi/src/test_concat.py", line 43, in wrapper
    raise TypeError('Argument {} must be {}'.format(name,bound_types[name]) )
TypeError: Argument name must be <class 'str'>
>>>
```

The `bind_partial` is used rather than `bind` so that we do not have to specify the type for all arguments; the idea behind the `partial` function from the `functools` module is also behind this.

The `inspect` module further defines some functions for interacting with classes and functions. A cross-section of these functions include:

1. `inspect.getclasstree(classes, unique=False)`: This arranges the list of classes into a hierarchy of nested lists. If the returned list contains a nested list, the nested list contains classes derived from the class whose entry immediately precedes the list. Each entry is a tuple containing a class and a tuple of its base classes.

```
>>> class Account:
...     pass
...
>>> class CheckingAccount(Account):
...     pass
...
>>> class SavingsAccount(Account):
...     pass
...
>>> import inspect
>>> inspect.getclasstree([Account, CheckingAccount, SavingsAccount])
[(<class 'object'>, ()), [(<class '__main__.Account'>, (<class 'object'>,)), [(<class '__main__.Chec\
kingAccount'>, (<class '__main__.Account'>,)), (<class '__main__.SavingsAccount'>, (<class '__main__\
.Account'>,))]]]
>>>
```

1. `inspect.getfullargspec(func)`: This returns the names and default values of a function's arguments; the return value is a named tuple of the form: `FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)`.
    - `args` is a list of the argument names.
    - `varargs` and `varkw` are the names of the `*` and `**` arguments or `None`.
    - `defaults` is an *n-tuple* of the default values of the last n arguments, or *None* if there are no default arguments.
    - *kwonlyargs* is a list of keyword-only argument names.
    - *kwonlydefaults* is a dictionary mapping names from kwonlyargs to defaults.
    - `annotations` is a dictionary mapping argument names to annotations.

    ```
    >>> def test(a:int, b:str='the') -> str:
    ...     pass
    ...
    >>> import inspect
    >>> inspect.getfullargspec(test)
    FullArgSpec(args=['a', 'b'], varargs=None, varkw=None, defaults=('the',), kwonlyargs=[], kwonl
    lts=None, annotations={'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'str'>})
    ```

2. `inspect.getargvalues(frame)`: This returns information about function arguments that have been passed into a particular frame. The return value is a named tuple `ArgInfo(args, varargs, keywords, locals)`.
    - `args` is a list of the argument names.
    - `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`.
    - `locals` is the *locals* dictionary of the given frame.
3. `inspect.getcallargs(func, *args, **kwds)`: This binds the `args` and `kwds` to the argument names of the function or method, `func`, as if it was called with them. For bound methods, bind also the first argument typically named `self` to the associated instance. A *dict* is returned, mapping the argument names including the names of the

`*` and `**` arguments, if any to their values from args and kwds. Whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised.

4. `inspect.getclosurevars(func)`: This returns the mapping of external name references in function or method, `func`, to their current values. A named tuple `ClosureVars(nonlocals, globals, builtins, unbound)` is returned.
    - `nonlocals` maps referenced names to lexical closure variables.
    - `globals` maps referenced names to the function's module globals and
    - `builtins` maps referenced names to the builtins visible from the function body.
    - `unbound` is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

The `inspect` module also supplies functions for accessing members of objects. An example of this is the `inspect.getmembers(object[, predicate])` that returns all attribute members of the `object` arguments; the predicate is an optional value that serves as a filter on the values returned. For example for a given class instance, `i`, we can get a list of attribute members of `i` that are methods by making the call `inspect.getmembers(i, inspect.ismethod)`; this returns a list of tuples of the attribute name and attribute object. The following example illustrates this.

```
    >>> acct = test_concat.Account("obi", 1000000000)
    >>> import inspect
    >>> inspect.getmembers(acct, inspect.ismethod)
    [('__getattr__', <bound method Account.__getattr__ of <test_concat.Account object at 0x101b3c4
>>), ('__init__', <bound method Account.__init__ of <test_concat.Account object at 0x101b3c470>>),
'del_account', <bound method Account.del_account of <test_concat.Account object at 0x101b3c470>>),
'deposit', <bound method Account.deposit of <test_concat.Account object at 0x101b3c470>>), ('inquir
', <bound method Account.inquiry of <test_concat.Account object at 0x101b3c470>>), ('withdraw', <bo
nd method Account.withdraw of <test_concat.Account object at 0x101b3c470>>)]
```

The `inspect` module has predicates for this method that include `isclass`, `ismethod`, `isfunction`, `isgeneratorfunction`, `isgenerator`, `istraceback`, `isframe`, `iscode`, `isbuiltin`, `isroutine`, `isabstract`, `ismethoddescriptor`.

10.3 Interacting with Interpreter Stacks

The `inspect` module also provides functions for dealing with interpeter stacks. The interpeter stack is composed of *frames*. All the functions below return a tuple of *the frame object*, *the filename*, *the line number of the current line*, *the function name*, *a list of lines of context from the source code*, and *the index of the current line within that list*. The provided functions enable user to inspect and manipulate the frame records.

1. `inspect.currentframe()`: This returns the frame object for the caller's stack frame. This function relies on stack frame support in the interpreter and this is not guaranteed to exist in all implementations of Python for example stackless python. If running in an implementation without Python stack frame support this function returns `None`.
2. `inspect.getframeinfo(frame, context=1)`: This returns information about the given argument frame or traceback object. A named tuple `Traceback(filename,`

`lineno, function, code_context, index)` is returned.

3. `inspect.getouterframes(frame, context=1)`: This returns a list of frame records for a given frame argument and all outer frames. These frames represent the calls that led to the creation of the argument frame. The first entry in the returned list represents the argument frame; the last entry represents the outermost call on the arugment frame's stack.
4. `inspect.getinnerframes(traceback, context=1)`: This returns a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of frame. The first entry in the list represents traceback; the last entry represents where the exception was raised.
5. `inspect.stack(context=1)`: This returns a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.
6. `inspect.trace(context=1)`: This returns a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

11. The Zen of Python …

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```