Contents

KARIM PICHARA – CHRISTIAN PIERINGER

ADVANCED COMPUTER PROGRAMMING IN PYTHON

ADVANCED COMPUTER PROGRAMMING IN PYTHON

ISBN: 9781521232385

To our wives and children

Preface

This book contains most of the relevant topics necessary to be an advanced computer programmer. The language used in the book is Python 3. Besides the programming language, the reader will learn most of the backbone contents in computer programming, such as object-oriented modeling, data structures, functional programming, input/output, simulation, graphical interfaces, and much more. We believe that the best way to learn computer programming is to work in hands-on activities. Practical exercises make the user get familiar with the main challenges that programming brings, such as how to model a particular problem; or how to write good code and efficient routine implementation, among others. Most of the chapters contain a set of hands-on activities (with proposed solutions) at the end. We encourage the readers to solve all those assignments without previously checking the solution. Challenges may be hard for initial programmers, but while going through this book, the activities will become more achievable for the reader. This book contains most of the material used for the *Advanced Python Programming* course taught at PUC University in Chile, by professors Karim Pichara and Christian Pieringer. The course is intended for Computer Science students as well as any other affine career that can be benefited by computer programming knowledge. Of course, this book is not enough to become a Software Engineer; there are other necessary courses that the reader must take to learn more advanced concepts related to the development of bigger software projects. Some of the recommended courses are *Database Systems, Data Structures, Operating Systems, Compilers, Software Engineering, Testing, Software Architecture*, and *Software Design*, among others. The content of this book will prepare the reader to have the necessary background for any of the next Software Engineering courses listed above. While using this book, readers should follow along on their computers to be able to try all the examples included in the chapters. It will be necessary that computers have already installed the required Python libraries.

Authors

Karim Pichara Baksai
Ph.D. in Computer Science,
Research Area: Machine Learning and Data Science applied to Astronomy
Associate Professor, Computer Science Department
Pontificia Universidad Católica de Chile (PUC)

Christian Pieringer Baeza
Ph.D. in Computer Science
Research Area: Computer Vision and Machine Learning
Adjunt Professor, Computer Science Department
Pontificia Universidad Católica de Chile (PUC)

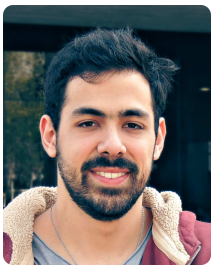
Acknowledgments

This book was not possible without the constant help of the teaching assistants; they gave us invaluable feedback, code and text editions to improve the book. The main collaborators who highly contributed are Belén Saldías, Ivania Donoso, Marco Bucchi, Patricio López, and Ignacio Becker.

We would like also thank the team of assistants who worked in the hands-on activities: Jaime Castro, Rodrigo Gómez, Bastián Mavrakis, Vicente Dominguez, Felipe Garrido, Javiera Astudillo, Antonio Gil, and José María De La Torre.

Belén Saldías

Ivania Donoso



Marco Bucci

Patricio López

Ignacio Becker

Chapter 1

Object Oriented Programming

In the real world, objects are tangible; we can touch and feel them, they represent something meaningful for us. In the software engineering field, objects are a virtual representation of entities that have a meaning within a particular context. In this sense, objects keep information/data related to what they represent and can perform actions/behaviors using their data. *Object Oriented Programming* (OOP) means that programs model functionalities through the interaction among objects using their data and behavior. The way OOP represents objects is an *abstraction*. It consists in to create a simplified model of the reality taking the more related elements according to the problem context and transforming them into attributes and behaviors. Assigning attributes and methods to objects involves two main concepts close related with the abstraction: *encapsulation* and *interface*.

Encapsulation refers to the idea of some attributes do not need to be visualized by other objects, so we can produce a cleaner code if we keep those attributes inside their respective object. For example, imagine we have the object `Amplifer` that includes attributes `tubes` and `power transformer`. These attributes only make sense inside the amplifier because other objects such as the `Guitar` do not need to interact with them nor visualize them. Hence, we should keep it inside the object `Amplifier`.

Interface let every object has a "facade" to protect its implementation (internal attributes and methods) and interact with the rest of objects. For example, an amplifier may be a very complex object with a bunch of electronic pieces inside. Think of another object such as the `Guitar player` and the `Guitar` that only interact with the amplifier through the input plug and knobs. Furthermore, two or more objects may have the same interface allowing us to replace them independently of their implementation and without change how we use them. Imagine a guitar player wants to try a tube amplifier and a solid state amp. In both cases, amplifiers have the interface (knobs an input plug) and offer the same user experience independently of their construction. In that sense, each object can provide the

suitable interface according to the context.

1.1 Classes

From the OOP perspective, classes describe objects, and each object is an instance of a class. The `class` statement allow us to define a class. For convention, we name classes using `CamelCase` and methods using `snake_case`. Here is an example of a class in Python:

```
# create_apartment.py


class Apartment:
    '''
    Class that represents an apartment for sale
    value is in USD
    '''

    def __init__(self, _id, mts2, value):
        self._id = _id
        self.mts2 = mts2
        self.value = value
        self.sold = False

    def sell(self):
        if not self.sold:
            self.sold = True
        else:
            print("Apartment {} was sold"
                  .format(self._id))
```

To create an object, we must create an instance of a class, for example, to create an apartment for sale we have to call the class `Apartment` with the necessary parameters to initialize it:

```
# instance_apartment.py

from create_apartment import Apartment
```

```
4
5  d1 = Apartment(_id=1, mts2=100, value=5000)
6
7  print("sold?", d1.sold)
8  d1.sell()
9  print("sold?", d1.sold)
10 d1.sell()
```

```
sold? False
sold? True
Apartment 1 was sold
```

We can see that the `__init__` method initializes the instance by setting the attributes (or data) to the initial values, passed as arguments. The first argument in the `__init__` method is `self`, which corresponds to the instance itself. Why do we need to receive the same instance as an argument? Because the `__init__` method is in charge of the initialization of the instance, hence it naturally needs access to it. For the same reason, every method defined in the class that specifies an action performed by the instance must receive `self` as the first argument. We may think of these methods as methods that belong to each instance. We can also define methods (inside a class) that are intended to perform actions within the class attributes, not to the instance attributes. Those methods belong to the class and do not need to receive `self` as an argument. We show some examples later.

Python provides us with the `help(<class>)` function to watch a description of a class:

```
1  help(Apartment)
```

```
#output

Help on class Apartment in module create_apartment:

class Apartment(builtins.object)
 |  Class that represents an apartment for sale
 |  price is in USD
 |
 |  Methods defined here:
 |
 |  __init__(self, _id, sqm, price)
```

```
 |
 |  sell(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

1.2 Properties

Encapsulation suggests some attributes and methods are private according to the object implementation, *i.e.*, they only exist within an object. Unlike other programming languages such as C++ or Java, in Python, the private concept does not exist. Therefore all attributes/methods are public, and any object can access them even if an interface exist. As a convention, we can suggest that an attribute or method to be private adding an underscore at the beginning of its name. For example, `_<attribute/method name>`. Even with this convention, we may access directly to the attributes or methods. We can strongly suggest that an element within an object is private using a double underscore `__<attribute/method name>`. The name of this approach is *name mangling*. It concerns to the fact of encoding addition semantic information into variables. Remember both approaches are conventions and good programming practices.

Properties are the pythonic mechanism to implement encapsulation and the interface to interact with private attributes of an object. It means every time we need that an attribute has a behavior we define it as property. In other way, we are forced to use a set of methods that allow us to change and retrieve the attribute values, e.g, the commonly used pattern `get_value()` and `set_value()`. This approach could generate us several maintenance problems.

The `property()` function allow us to create a property, receiving as arguments the functions use to get, set and delete the attribute as `property(<setter_function>, <getter_function>, <deleter_function>)`. The next example shows the way to create a property:

```
1  # property.py
2
3  class Email:
```

```
4
5       def __init__(self, address):
6           self._email = address  # A private attribute
7
8       def _set_email(self, value):
9           if '@' not in value:
10              print("This is not an email address.")
11          else:
12              self._email = value
13
14      def _get_email(self):
15          return self._email
16
17      def _del_email(self):
18          print("Erase this email attribute!!")
19          del self._email
20
21      # The interface provides the public attribute email
22      email = property(_get_email, _set_email, _del_email,
23                  'This property contains the email.')
```

Check out how the property works once we create an instance of the `Email` class:

```
m1 = Email("kp1@othermail.com")
print(m1.email)
m1.email = "kp2@othermail.com"
print(m1.email)
m1.email = "kp2.com"
del m1.email
```

```
kp1@othermail.com
kp2@othermail.com
This is not an email address.
Erase this email attribute!!
```

Note that properties makes the assignment of internal attributes easier to write and read. Python also let us to define properties using decorators. Decorators is an approach to change the behavior of a method. The way to create a

property through decorators is adding `@property` statement before the method we want to define as attribute. We explain decorators in Chapter 3.

```
# property_without_decorator.py

class Color:

    def __init__(self, rgb_code, name):
        self.rgb_code = rgb_code
        self._name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name

    name = property(get_name, set_name)
```

```
# property_with_decorator.py

class Color:

    def __init__(self, rgb_code, name):
        self._rgb_code = rgb_code
        self._name = name

    # Create the property using the name of the attribute. Then we
    # define how to get/set/delet it.
    @property
    def name(self):
        print("Function to get the name color")
        return self._name

    @name.setter
    def name(self, new_name):
```

```
18          print("Function to set the name as {}".format(new_name))
19          self._name = new_name
20
21      @name.deleter
22      def name(self):
23          print("Erase the name!!")
24          del self._name
```

1.3 Aggregation and Composition

In OOP there are different ways from which objects interact. Some objects are a composition of other objects who only exists for that purpose. For instance, the object `printed circuit board` only exists inside a `amplifier` and its existence only last while the `amplifier` exists. That kind of relationship is called *composition*. Another kind of relationship between objects is *aggregation*, where a set of objects compose another object, but they may continue existing even if the composed object no longer exist. For example, students and a teacher compose a classroom, but both are not meant to be just part of that classroom, they may continue existing and interacting with other objects even if that particular classroom disappears. In general, *aggregation* and *composition* concepts are different from the modeling perspective. The use of them depends on the context and the problem abstraction. In Python, we can see *aggregation* when the composed object receive instances of the components as arguments, while in *composition*, the composed object instantiates the components at its initialization stage.

1.4 Inheritance

The inheritance concept allows us to model relationships like "object B is an object A but specialized in certain functions". We can see a subclass as a specialization of its superclass. For example, lets say we have a class called `Car` which has attributes: `brand, model` and `year`; and methods: `stop`, `charge_gas` and `fill_tires`. Assume that someone asks us to model a taxi, which is a car but has some additional specifications. Since we already have defined the `Car` class, it makes sense somehow to re-use its attributes and methods to create a new `Taxi` class (subclass). Of course, we have to add some specific attributes and methods to `Taxi`, like `taximeter, fares` or `create_receipt`. However, if we do not take advantage of the `Car` superclass by inheriting from it, we will have to repeat a lot of code. It makes our software much harder to maintain.

Besides inheriting attributes and methods from a superclass, inheritance allows us to "re-write" superclass methods. Suppose that the subclass `Motorcycle` inherits from the class `Vehicle`. The method called `fill_tires` from

`Vehicle` has to be changed inside `Motorcycle`, because motorcycles (in general) have two wheels instead of four. In Python, to modify a method in the subclass we just need to write it again, this is called *overriding*, so that Python understands that every time the last version of the method is the one that holds for the rest of the code.

A very useful application of inheritance is to create subclasses that inherit from some of the Python built-in classes, to extend them into a more specialized class. For example, if we want to create a custom class similar to the built-in class `list`, we just must create a subclass that inherits from `list` and write the new methods we want to add:

```
1  # grocery_list.py
2
3
4  class GroceryList(list):
5
6      def discard(self, price):
7          for product in self:
8              if product.price > price:
9                  # remove method is implemented in the class "list"
10                 self.remove(product)
11         return self
12
13     def __str__(self):
14         out = "Grocery List:\n\n"
15         for p in self:
16             out += "name: " + p.name + " - price: "
17             + str(p.price) + "\n"
18
19         return out
20
21
22 class Product:
23
24     def __init__(self, name, price):
25         self.name = name
26         self.price = price
27
28
```

```
29 grocery_list = GroceryList()
30
31 # extend method also belongs to 'list' class
32 grocery_list.extend([Product("bread", 5),
33                      Product("milk", 10), Product("rice", 12)])
34
35 print(grocery_list)
36 grocery_list.discard(11)
37 print(grocery_list)
```

```
Grocery List:

name: bread - price: 5
name: milk - price: 10
name: rice - price: 12

Grocery List:

name: bread - price: 5
name: milk - price: 10
```

Note that the `__str__` method makes the class instance able to be printed out, in other words, if we call `print(grocery_list)` it will print out the string returned by the `__str__` method.

1.5 Multiple Inheritance

We can inherit from more than one class. For example, a professor might be a teacher and a researcher, so she/he should inherit attributes and methods from both classes:

```
1 # multiple_inheritance.py
2
3
4 class Researcher:
5
6     def __init__(self, field):
7         self.field = field
```

```
8
9       def __str__(self):
10          return "Research field: " + self.field + "\n"
11
12
13  class Teacher:
14
15      def __init__(self, courses_list):
16          self.courses_list = courses_list
17
18      def __str__(self):
19          out = "Courses: "
20          for c in self.courses_list:
21              out += c + ", "
22          # the [:-2] selects all the elements
23          # but the last two
24          return out[:-2] + "\n"
25
26
27  class Professor(Teacher, Researcher):
28
29      def __init__(self, name, field, courses_list):
30          # This is not completetly right
31          # Soon we will see why
32          Researcher.__init__(self, field)
33          Teacher.__init__(self, courses_list)
34          self.name = name
35
36      def __str__(self):
37          out = Researcher.__str__(self)
38          out += Teacher.__str__(self)
39          out += "Name: " + self.name + "\n"
40          return out
41
42
```

```
43 p = Professor("Steve Iams",
44                "Meachine Learning",
45                [
46                  "Python Programming",
47                  "Probabilistic Graphical Models",
48                  "Bayesian Inference"
49                ])
50
51 print(p)

   #output

   Research field: Meachine Learning
   Courses: Python Programming, Probabilistic Graphical Models,
            Bayesian Inference
   Name: Steve Iams
```

Multiple Inheritance Problems

In Python, every class inherits from the `Object` class, that means, among other things, that every time we instantiate a class, we are indirectly creating an instance of `Object`. Assume we have a class that inherits from several superclasses. If we call to all the `__init__` superclass methods, as we did in the previous example (calling `Researcher.__init__` and `Teacher.__init__`), we are calling the `Object` initializer twice: `Researcher.__init__` calls the initialization of `Object` and `Teacher.__init__` calls the initialization of `Object` as well. Initializing objects twice is not recommended. It is a waste of resources, especially in cases where the initialization is expensive. It could be even worst. Imagine that the second initialization changes the setup done by the first one, and probably the objects will not notice it. This situation is known as *The Diamond Problem*.

The following example (taken from [6]) shows what happens in the context of multiple-inheritance if each subclass calls directly to initialize all its superclasses. Figure 1.1 indicates the hierarchy of the classes involved.

The example below (taken from [6]) shows what happens when we call the `call()` method in both superclasses from `SubClassA`.

```
1 # diamond_problem.py
2
```

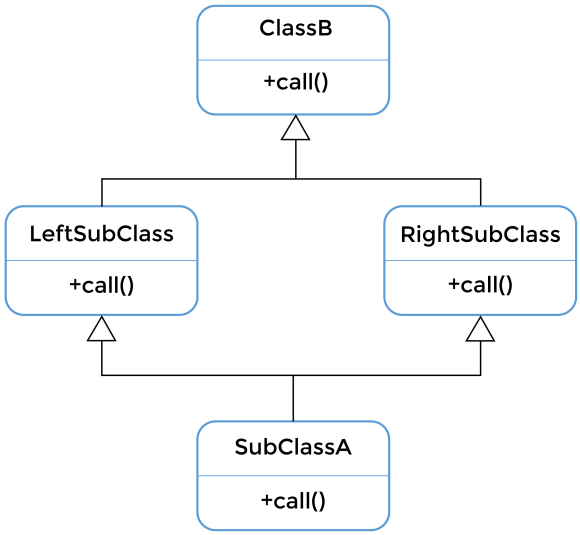


Figure 1.1: The diamond problem

```
3  class ClassB:
4      num_calls_B = 0
5
6      def make_a_call(self):
7          print("Calling method in ClassB")
8          self.num_calls_B += 1
9
10
11 class LeftSubClass(ClassB):
12     num_left_calls = 0
13
14     def make_a_call(self):
15         ClassB.make_a_call(self)
16         print("Calling method in LeftSubClass")
17         self.num_left_calls += 1
18
19
20 class RightSubClass(ClassB):
21     num_right_calls = 0
```

```
22
23      def make_a_call(self):
24          ClassB.make_a_call(self)
25          print("Calling method in RightSubClass")
26          self.num_right_calls += 1
27
28
29  class SubClassA(LeftSubClass, RightSubClass):
30      num_calls_subA = 0
31
32      def make_a_call(self):
33          LeftSubClass.make_a_call(self)
34          RightSubClass.make_a_call(self)
35          print("Calling method in SubClassA")
36          self.num_calls_subA += 1
37
38
39  if __name__ == '__main__':
40      s = SubClassA()
41      s.make_a_call()
42      print("SubClassA: {}".format(s.num_calls_subA))
43      print("LeftSubClass: {}".format(s.num_left_calls))
44      print("RightSubClass: {}".format(s.num_right_calls))
45      print("ClassB: {}".format(s.num_calls_B))
```

```
Calling method in ClassB
Calling method in LeftSubClass
Calling method in ClassB
Calling method in RightSubClass
Calling method in SubClassA
SubClassA: 1
LeftSubClass: 1
RightSubClass: 1
ClassB: 2
```

From the output, we can see that the upper class in the hierarchy (`ClassB`) is called twice, despite that we just directly

call it once in the code.

Every time that one class inherits from two classes, a diamond structure is created. We refer the readers to https://www.python.org/doc/newstyle/ for more details about *new style classes*. Following the same example shown above, if instead of calling the `make_a_call()` method we call the `__init__` method, we would be initializing `Object` twice.

Solution to the diamond problem

One possible solution to the diamond problem is that each class must call the initialization of the superclass that precedes it in the multiple inheritance resolution order. In Python, the order goes from left to right on the list of superclasses from where the subclass inherits.

In this case, we just should call to `super()`, because Python will make sure to call the parent class in the multiple inheritance resolution order. In the previous example, after the subclass goes `LeftSubclass`, then `RightSubClass`, and finally `ClassB`. From the following output, we can see that each class was initialized once:

```
1  # diamond_problem_solution.py
2
3  class ClassB:
4      num_calls_B = 0
5
6      def make_a_call(self):
7          print("Calling method in ClassB")
8          self.num_calls_B += 1
9
10
11 class LeftSubClass(ClassB):
12     num_left_calls = 0
13
14     def make_a_call(self):
15         super().make_a_call()
16         print("Calling method in LeftSubClass")
17         self.num_left_calls += 1
18
19
```

```
20 class RightSubClass(ClassB):
21     num_right_calls = 0
22
23     def make_a_call(self):
24         super().make_a_call()
25         print("Calling method in RightSubClass")
26         self.num_right_calls += 1
27
28
29 class SubClassA(LeftSubClass, RightSubClass):
30     num_calls_subA = 0
31
32     def make_a_call(self):
33         super().make_a_call()
34         print("Calling method in SubClassA")
35         self.num_calls_subA += 1
36
37 if __name__ == '__main__':
38
39     s = SubClassA()
40     s.make_a_call()
41     print("SubClassA: {}".format(s.num_calls_subA))
42     print("LeftSubClass: {}".format(s.num_left_calls))
43     print("RightSubClass: {}".format(s.num_right_calls))
44     print("ClassB: {}".format(s.num_calls_B))
```

```
Calling method in ClassB
Calling method in RightSubClass
Calling method in LeftSubClass
Calling method in SubClassA
SubClassA: 1
LeftSubClass: 1
RightSubClass: 1
ClassB: 1
```

Method Resolution Order

The `mro` method shows us the hierarchy order. It is very useful in more complex multiple-inheritance cases. Python uses the C3 [3] algorithm to calculate a linear order among the classes involved in multiple inheritance schemes.

```
1  # mro.py
2
3  from diamond_problem_solution import SubClassA
4
5  for c in SubClassA.__mro__:
6      print(c)
```

```
<class 'diamond_problem_solution.SubClassA'>
<class 'diamond_problem_solution.LeftSubClass'>
<class 'diamond_problem_solution.RightSubClass'>
<class 'diamond_problem_solution.ClassB'>
<class 'object'>
```

The next example describes a case of an unrecommended initialization. The C3 algorithm generates an error because it cannot create a logical order:

```
1  # invalid_structure.py
2
3  class X():
4      def call_me(self):
5          print("I'm X")
6
7
8  class Y():
9      def call_me(self):
10         print("I'm Y")
11
12
13 class A(X, Y):
14     def call_me(self):
15         print("I'm A")
16
```

```
17
18 class B(Y, X):
19     def call_me(self):
20         print("I'm B")
21
22
23 class F(A, B):
24     def call_me(self):
25         print("I'm F")
26
27 # TypeError: Cannot create a consistent method resolution
28 # order (MRO) for bases X, Y
```

```
Traceback (most recent call last):
  File "/codes/invalid_structure.py",
line 24, in <module>
    class F(A, B):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases X, Y
```

A Multiple Inheritance Example

Here we present another case of multiple-inheritance, showing the wrong and the right way to call the initialization of superclasses:

Wrong initialization of the superclass' `__init__` method

Calling directly to superclasses' `__init__` method inside the class `Customer`, as we show in the next example, is highly not recommended. We could initiate a superclass multiple times, as we mentioned previously. In this example, a call to `object`'s `__init__` is done twice:

```
1 # inheritance_wrong.py
2
3
4 class AddressHolder:
5
6     def __init__(self, street, number, city, state):
```

```
7           self.street = street
8           self.number = number
9           self.city = city
10          self.state = state
11
12
13  class Contact:
14
15      contact_list = []
16
17      def __init__(self, name, email):
18          self.name = name
19          self.email = email
20          Contact.contact_list.append(self)
21
22
23  class Customer(Contact, AddressHolder):
24
25      def __init__(self, name, email, phone,
26                   street, number, state, city):
27          Contact.__init__(self, name, email)
28          AddressHolder.__init__(self, street, number,
29                                 state, city)
30          self.phone = phone
31
32
33  if __name__ == "__main__":
34
35      c = Customer('John Davis', 'jp@g_mail.com', '23542331',
36                   'Beacon Street', '231', 'Cambridge', 'Massachussets')
37
38      print("name: {}\nemail: {}\naddress: {}, {}"
39            .format(c.name, c.email, c.street, c.state))
```

```
name: John Davis
email: jp@g_mail.com
```

```
address: Beacon Street, Massachussets
```

The right way: *args y **kwargs

Before showing the fixed version of the above example, we show how to use a list of arguments (`*args`) and keyword arguments (`**kwargs`). In this case `*args` refers to a *Non-keyword variable length argument list*, where the operator `*` unpacks the content inside the list `args` and pass them to a function as **positional** arguments.

```
1  # args_example.py
2
3  def method2(f_arg, *argv):
4      print("first arg normal: {}".format(f_arg))
5      for arg in argv:
6          print("the next arg is: {}".format(arg))
7
8  if __name__ == "__main__":
9      method2("Lorem", "ipsum", "ad", "his", "scripta")
```

```
first arg normal: Lorem
the next arg is: ipsum
the next arg is: ad
the next arg is: his
the next arg is: scripta
```

Similarly, `**kwargs` refers to a *keyword variable-length argument list*, where `**` maps all the elements within the dictionary `kwargs` and pass them to a function as **non-positional** arguments. This method is used to send a variable amount of arguments to a function:

```
1  # kwargs_example.py
2
3  def method(arg1, arg2, arg3):
4      print("arg1: {}".format(arg1))
5      print("arg2: {}".format(arg2))
6      print("arg3: {}".format(arg3))
7
8
```

```
9  if __name__ == "__main__":
10     kwargs = {"arg3": 3, "arg2": "two"}
11     method(1, **kwargs)
```

```
arg1: 1
arg2: two
arg3: 3
```

Now that we know how to use `*args` and `**kwargs`, we can figure out how to properly write an example of multiple inheritance as shown before:

```
1  # inheritance_right.py
2
3
4  class AddressHolder:
5
6      def __init__(self, street, number, city, state, **kwargs):
7          super().__init__(**kwargs)
8          self.street = street
9          self.number = number
10         self.city = city
11         self.state = state
12
13
14 class Contact:
15     contact_list = []
16
17     def __init__(self, name, email, **kwargs):
18         super().__init__(**kwargs)
19         self.name = name
20         self.email = email
21         Contact.contact_list.append(self)
22
23
24 class Customer(Contact, AddressHolder):
25
```

```
26      def __init__(self, phone_number, **kwargs):
27          super().__init__(**kwargs)
28          self.phone_number = phone_number
29
30
31  if __name__ == "__main__":
32
33      c = Customer(name='John Davis', email='jp@g_mail.com',
34                   phone_number='23542331', street='Beacon Street',
35                   number='231', city='Cambridge', state='Massachussets')
36
37      print("name: {}\nemail: {}\naddress: {}, {}".format(c.name, c.email,
38                                                          c.street, c.state))
```

```
name: John Davis
email: jp@g_mail.com
address: Beacon Street, Massachussets
```

As we can see in the above example, each class manage its own arguments passing the rest of the non-used arguments to the higher classes in the hierarchy. For example, `Customer` passes all the non-used argument (`**args`) to `Contact` and to `AddressHolder` through the `super()` function.

Polymorfism

Imagine that we have the `ChessPiece` class. This class has six subclasses: `King`, `Queen`, `Rook`, `Bishop`, `Knight`, and `Pawn`. Each subclass contains the `move` method, but that method behaves differently on each subclass. The ability to call a method with the same name but with different behavior within subclasses is called *Polymorphism*. There are mainly two flavors of polymorphism:

- *Overriding*: occurs when a subclass implements a method that replaces the same method previously implemented in the superclass.
- *Overloading*: happens when a method is implemented more than once, having a different number of arguments on each case. Python does not support overloading because it is not really necessary. Each method can have a variable number of arguments by using a keyworded o non-keyworded list of arguments. Recall that in Python every time we implement a method more than once, the last version is the only one that Python will use. Other

programming languages support overloading, automatically detecting what method implementation to use depending on the number of present arguments when the method is called.

The code bellow shows an example of Overriding. The `Variable` class represents data of any kind. The `Income` class contains a method to calculate the representative value for it. In `Income`, the representative value is the average, in `City`, the representative value is the most frequent city, and in `JobTitle`, the representative value is the job with highest range, according to the `_range` dictionary:

```
# polymorfism_1.py

class Variable:
    def __init__(self, data):
        self.data = data

    def representative(self):
        pass


class Income(Variable):
    def representative(self):
        return sum(self.data) / len(self.data)


class City(Variable):
    # class variable
    _city_pop_size = {'Shanghai': 24000, 'Sao Paulo': 21300, 'Paris': 10800,
                      'London': 8600, 'Istambul': 15000,
                      'Tokyo': 13500, 'Moscow': 12200}

    def representative(self):
        dict = {City._city_pop_size[c]: c for c in self.data
                if c in City._city_pop_size.keys()}
        return dict[max(dict.keys())]


class JobTitle(Variable):
```

```
29      # class variable
30      _range = {'CEO': 1, 'CTO': 2, 'Analyst': 3, 'Intern': 4}
31
32      def representative(self):
33          dict = {JobTitle._range[c]: c for c in self.data if
34                  c in JobTitle._range.keys()}
35          return dict[min(dict.keys())]
36
37
38  if __name__ == "__main__":
39      income_list = Income([50, 80, 90, 150, 45, 65, 78, 89, 59, 77, 90])
40      city_list = City(['Shanghai', 'Sao Paulo', 'Paris', 'London',
41                        'Istambul', 'Tokyo', 'Moscow'])
42      job_title_list = JobTitle(['CTO', 'Analyst', 'CEO', 'Intern'])
43      print(income_list.representative())
44      print(city_list.representative())
45      print(job_title_list.representative())
```

```
79.36363636363636
Shanghai
CEO
```

Operator Overriding

Python has several built-in operators that work for many of the built-in classes. For example, the operator "+" can sum up two numbers, concatenate two strings, mix two lists, etc., depending on the object we are working with. The following code shows an example:

```
1  # operator_overriding_1.py
2
3  a = [1, 2, 3, 4]
4  b = [5, 6, 7, 8]
5  print(a + b)
6
7  c = "Hello"
8  d = " World"
9  print(c + d)
```

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

Thanks to polymorphism, we can also personalize the method `__add__` to make it work on any particular class we want. For example, we may need to create a specific way of adding two instances of the `ShoppingCart` class in the following code:

```
# operator_overriding_2.py

class ShoppingCart:

    def __init__(self, product_list):
        self.product_list = product_list  # Python dictionary

    def __call__(self, product_list = None):
        if product_list is None:
            product_list = self.product_list
        self.product_list = product_list

    def __add__(self, other_cart):
        added_list = self.product_list
        for p in other_cart.product_list.keys():
            if p in self.product_list.keys():
                value = other_cart.product_list[p] + self.product_list[p]
                added_list.update({p: value})
            else:
                added_list.update({p: other_cart.product_list[p]})

        return ShoppingCart(added_list)

    def __repr__(self):
        return "\n".join("Product: {} | Quantity: {}".format(
            p, self.product_list[p]) for p in self.product_list.keys()
        )


```

```
30 if __name__ == "__main__":
31     s_cart_1 = ShoppingCart({'muffin': 3, 'milk': 2, 'water': 6})
32     s_cart_2 = ShoppingCart({'milk': 5, 'soda': 2, 'beer': 12})
33     s_cart_3 = s_cart_1 + s_cart_2
34     print(s_cart_3.product_list)
35     print(s_cart_3)
```

```
{'soda': 2, 'water': 6, 'milk': 7, 'beer': 12, 'muffin': 3}
Product: soda | Quantity: 2
Product: water | Quantity: 6
Product: milk | Quantity: 7
Product: beer | Quantity: 12
Product: muffin | Quantity: 3
```

The `__repr__` method allows us to generate a string that will be used everytime we ask to `print` any instance of the class. We could also implement the `__str__` method instead, it works almost exactly as `__repr__`, the main difference is that `__str__` should be used when we need a user friendly print of the object instance, related to the particular context where it is used. The `__repr__` method should generate a more detailed printing, containing all the necessary information to understand the instance. In cases where `__str__` and `__repr__` are both implemented, Python will use `__str__` when `print(instance)` is called. Hence, `__repr__` will be used only if `__str__` is not implemented.

There are other operators that we can override, for example "less than" (`__lt__`), "greather than" (`__gt__`) and "equal" (`__eq__`). We refer the reader to `https://docs.python.org/3.4/library/operator.html` for a detailed list of built-in operators. Here is an example that shows how to override the `__lt__` method for implementing the comparison between two elements of the `Point` class:

```
1 # operator_overriding_3.py
2
3 class Point:
4
5     def __init__(self, x, y):
6         self.x = x
7         self.y = y
8
9     def __lt__(self, other_point):
```

```
10          self_mag = (self.x ** 2) + (self.y ** 2)
11          other_point_mag = (other_point.x ** 2) + (other_point.y ** 2)
12          return self_mag < other_point_mag
13
14  if __name__ == "__main__":
15      p1 = Point(2, 4)
16      p2 = Point(8, 3)
17      print(p1 < p2)

    True
```

Duck Typing

The most common way to define it is "If it walks like a duck and quacks like a duck, then it is a duck." In other words, it does not matter what kind of object performs the action, if it can do it, let it do it. Duck typing is a feature that some programming languages have that makes polymorphism less attractive because it allows polymorphic behavior without inheritance. In the next example, we can see that the `activate` function makes a duck `scream` and `walk`. Despite that the method is implemented for `Duck` objects, it can be used with any object that has the `scream` and `walk` methods implemented:

```
1   # duck_typing.py
2
3
4   class Duck:
5
6       def scream(self):
7           print("Cuack!")
8
9       def walk(self):
10          print("Walking like a duck...")
11
12
13  class Person:
14
15      def scream(self):
16          print("Ahhh!")
```

```
17
18      def walk(self):
19          print("Walking like a human...")
20
21
22  def activate(duck):
23      duck.scream()
24      duck.walk()
25
26  if __name__ == "__main__":
27      Donald = Duck()
28      John = Person()
29      activate(Donald)
30      # this is not supported in other languajes, because John
31      # is not a Duck object
32      activate(John)
```

```
Cuack!
Walking like a duck...
Ahhh!
Walking like a human...
```

Typed programming languages that verify the type of objects during compilation time, such as C/C++, do not support duck typing because in the list of arguments the object's type has to be specified.

1.6 Abstract Base Class

Abstract classes in a programming language allow us to represent abstract objects better. What is an abstract object? Abstract objects are created only to be inherited, so it does not make any sense to instantiate them. For example, imagine that we have cars, buses, ships, and planes. Each one has similar properties (passengers capacity, color, among others) and actions (load, move, stop) but they implement their methods differently: it is not the same to stop a ship than a car. For this reason, we can create the class called `Vehicle`, to be a superclass of `Car`, `Bus`, `Ship` and `Plane`.

Our problem now is how do we define `Vehicle`'s actions. We cannot define any of those behaviors inside `Vehicle`. We can only do it inside any of its subclasses. That explains why it does not make any sense to instantiate the abstract

class `Vehicle`. Hence, it is recommended to make sure that abstract classes are not going to be instantiated in the code, by raising an exception in case any programmer attempts to instantiate it.

We can also have abstract methods, in other words, methods that have to be defined in the subclasses because their implementation makes no sense to occur in the abstract class. Abstract classes can also have traditional (non abstract) methods when they do not need to be modified withing the subclasses. Let's try to define abstract classes and abstract methods in Python. The following code shows an example:

```
1  # 01_abstract_1.py
2
3  class Base:
4      def func_1(self):
5          raise NotImplementedError()
6
7      def func_2(self):
8          raise NotImplementedError()
9
10 class SubClass(Base):
11     def func_1(self):
12         print("func_1() called...")
13         return
14
15     # We intentionally did not implement func_2
16
17 b1 = Base()
18 b2 = SubClass()
19 b2.func_1()
20 b2.func_2()
```

```
func_1() called...
---------------------------------------------------
NotImplementedError Traceback (most recent call last)
<ipython-input-17-0803174cce17> in <module>()
     16 b2 = SubClass()
     17 b2.func_1()
---> 18 b2.func_2()
```

```
<ipython-input-17-0803174cce17> in func_2(self)
      4
      5     def func_2(self):
----> 6         raise NotImplementedError()
      7
      8 class SubClass(Base):

NotImplementedError:
```

The problem with this approach is that the program lets us instantiate `Base` without complaining, that is not what we want. It also allows us not to implement all the needed methods in the subclass. An Abstract class allows the class designer to assert that the user will implement all the required methods in the respective subclasses.

Python, unlike other programming languages, does not have a built-in way to declare abstract classes. Fortunately, we can import the `ABC` module (stands for Abstract Base Class), that satisfies our requirements. The following code shows an example:

```
1  # 03_ABC_1.py
2
3  from abc import ABCMeta, abstractmethod
4
5
6  class Base(metaclass=ABCMeta):
7      @abstractmethod
8      def func_1(self):
9          pass
10
11     @abstractmethod
12     def func_2(self):
13         pass
14
15
16 class SubClass(Base):
17     def func_1(self):
18         pass
19
```

```
20      # We intentionally did not implement func_2
21
22  print('Is it subclass?: {}'.format(issubclass(SubClass, Base)))
23  print('Is it instance?: {}'.format(isinstance(SubClass(), Base)))
```

```
Is it subclass?: True
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-19-b1003dd6fd92> in <module>()
     17
     18 print('Is it subclass?: {}'.format(issubclass(SubClass, Base)))
---> 19 print('Is it instance?: {}'.format(isinstance(SubClass(), Base)))

TypeError: Can't instantiate abstract class SubClass with abstract methods ' \
              'func_2
```

```
1  # 04_ABC_2.py
2
3  print('Trying to generate an instance of the Base class\n')
4  a = Base()
```

```
Trying to generate an instance of the Base class
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-20-e8aa694c9937> in <module>()
      1 print('Trying to generate an instance of the Base class\n')
----> 2 a = Base()

TypeError: Can't instantiate abstract class Base with abstract methods
              func_1, func_2
```

Note that to declare a method as abstract we have to add the `@abstractmethod` decorator over its declaration. The following code shows the implementation of the abstract methods:

```
1  # 05_ABC_3.py
2
3  from abc import ABCMeta, abstractmethod
```

```
4
5
6  class Base(metaclass=ABCMeta):
7      @abstractmethod
8      def func_1(self):
9          pass
10
11     @abstractmethod
12     def func_2(self):
13         pass
14
15
16 class SubClass(Base):
17
18     def func_1(self):
19         pass
20
21     def func_2(self):
22         pass
23
24     # We forgot again to implement func_2
25
26 c = SubClass()
27 print('Subclass: {}'.format(issubclass(SubClass, Base)))
28 print('Instance: {}'.format(isinstance(SubClass(), Base)))
```

```
Subclass: True
Instance: True
```

1.7 Class Diagrams

By using class diagrams, we can visualize the classes that form most of the objects in our problem. Besides the classes, we can also represent their properties, methods, and the way other classes interact with them. These diagrams belong to a language known as *Unified Modelling Language* (UML). UML allows us to incorporate elements and tools to model more complex systems, but it is out of the scope of this book. A class diagram is composed of a set of classes

and their relations. Rectangles represent classes, and they have three fields; the class name, its data (attributes and variables), and its methods. Figure 1.2 shows an example.

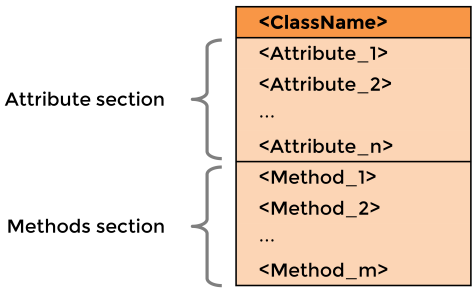


Figure 1.2: UML Class diagram example

Suppose that by using OOP we want to model a database with stars that exist in a given area of the Universe. Each star is correspond to a set of T observations, where each observation is a tuple (m_i, t_i, e_i), where m_i is the bright magnitude of the star in observation i, t_i is the time when the observation i was obtained, and e_i is the instrumental error associated with observation i. Using UML diagrams, we can model the system as shown in figure 1.3.

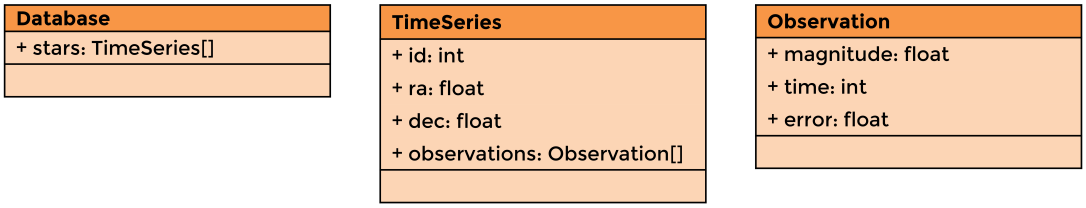


Figure 1.3: UML tables for the stars model

Consider now that the `TimeSeries` class has methods to add an observation to the time series and to return the average and standard deviation of the set of observations that belong to the star. We can represent the class and the mentioned methods as in figure 1.4.

Besides the classes, we can represent the relations among them using UML notation as well. The most common types of relations are: *composition*, *agreggation* and *inheritance* (Concepts explained previously in this chapter). Figure 1.5 shows an example of *Composition*. This relation is represented by an arrow starting from the base object towards the target that the class is composing. The base of the connector is a black diamond. The number at the beginning and end of the arrow represent the *cardinalities*, in other words, the range of the number of objects included in each side of the relation. In this example, the number one at the beginning of the arrow and the $1, \ldots, *$ at the end, mean that *one* Time Series may include a set of *one* or more observations, respectively.

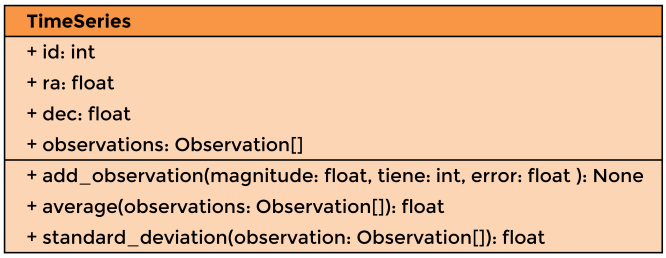


Figure 1.4: UML table for the `TimeSeries` class including the `add_observation`, `average` and `standard_deviation` methods.

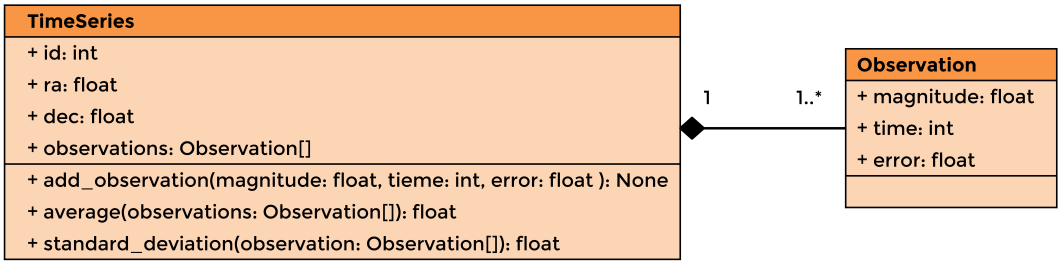


Figure 1.5: Example of the *composition* relation between classes in a UML diagram.

Similarly, *agreggation* is represented as an arrow with a hollow diamond at the base. Figure 1.6 shows an example. As we learn previously in this chapter, here the only difference is that the database aggregates time series, but the time series objects are entities that have a significance even out of the database where they are aggregate.

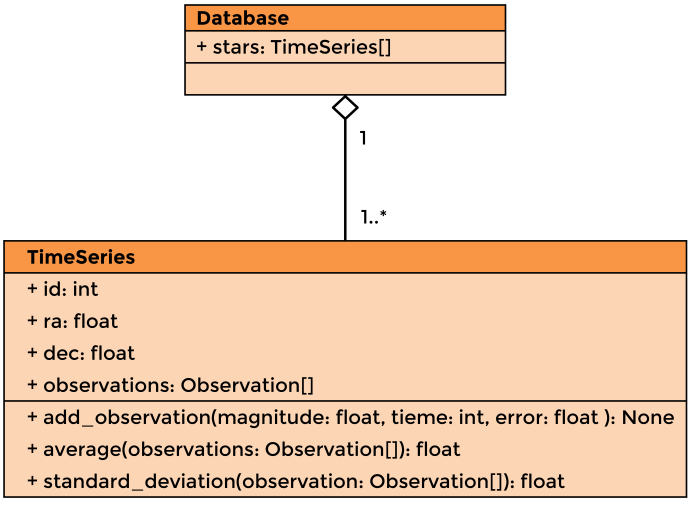


Figure 1.6: Example of the *agreggation* relation between classes in a UML diagram.

In UML, we represent the *inheritance* as a simple arrow with a hollow triangle at the head. To show an example, consider the case of astronomical stars again. Imagine that some of the time series are `periodic`. It means that there is a set of values repeated at a constant amount of time. According to that, we can define the subclass `PeriodicTimeSeries` that inherits from the `TimeSeries` class its common attributes and behaviors. Figure 1.7 shows this example in a UML diagram. The complete UML diagram for the astronomical stars example is shown in figure 1.8.

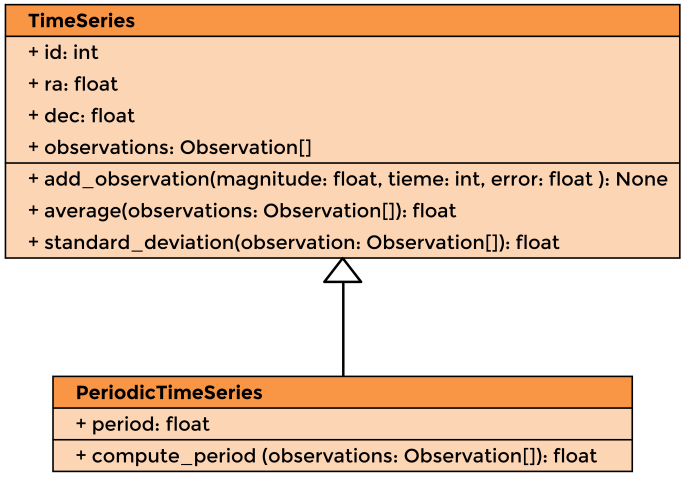


Figure 1.7: Example of the *inheritance* relation between classes in a UML diagram.

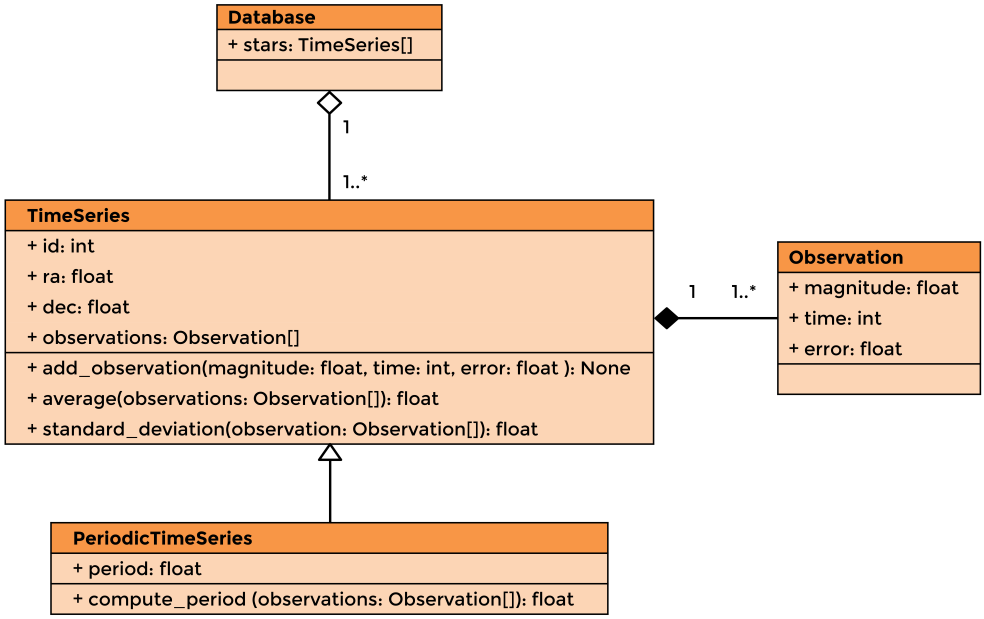


Figure 1.8: The complete UML diagram for the astronomical stars example.

1.8 Hands-On Activities

Activity 1.1

Variable stars are stars whose level of brightness changes with time. They are of considerable interest in the astronomical community. Each star belongs to one of the possible classes of variability. Some of the classes are RR Lyrae, Eclipsing Binaries, Mira, Long Period Variables, Cepheids, and Quasars. Each star has a position in the sky represented by RA and DEC coordinates. Stars are represented by an identifier and contain a set of observations. Each observation is a tuple with three values: time, magnitude and error. Time value indicates the moment in which the telescope or the instrument does the observation. The magnitude indicates the amount of brightness calculated in the observation. The error corresponds to a range of uncertainty associated with the measurement. Many fields compose the sky, and each field contains a huge amount of stars. For each star, we need to know the average and the variance of the bright magnitudes. Create the necessary Python classes which allow modeling the situation described above. Classes must have the corresponding attributes and methods. Write code to instantiate the classes involved.

Activity 1.2

A software company is building a computer program that works with geometric shapes and needs help with the initial model. The company is interested in making the model as extensible as possible.

- **Each figure has:**
  - A property `center` as an ordered pair `(x,y)`. It should be possible to access and set it. The constructor needs one to build a figure.
  - A method `translate` that receives an ordered pair `(a, b)` and sums to each of the center's component, the components in `(a, b)`.
  - A property `perimeter` that should be calculated with the figure's dimensions.
  - A property `area` that should be calculated with the figure's dimensions.
  - A method `grow_area` that enlarges the area of the figure x times, increasing its dimensions proportionally. For example, in the case of a rectangle it should modify its `width` y `length`.
  - A method `grow_perimeter` that enlarges the perimeter in x units, increasing its dimensions proportionally. For example, in the case of a rectangle it should modify its `width` y `length`.
  - Each time a figure is printed, the output should have the following format:
    `ClassName - Perimeter: value, Area: value, Center: (x, y)`

- **A rectangle has:**
  - Two properties, `length` and `width`. It must be possible to access and set them. The constructor needs both of them to build a figure.
- **An Equilateral Triangle has:**
  - A property `side` that must be possible to access and to set. The constructor needs it to build a figure.

Implement all the necessary classes to represent the model proposed. Some of the classes and methods might be abstract. Also implement a `vertices` property for all the figures that returns a list of ordered pairs `(x, y)`. **Hint:** Recall that to change what `print(object)` shows, you have to override the method `__repr__`.

Chapter 2

Data Structures

We define a **data structure** as a specific way to group and manage the information, such that we can efficiently use the data. Opposite to the simple variables, a data structure is an abstract data type that involves a high level of abstraction, and therefore a tight relation with *OOP*. We will show the Python implementation of every data structure according to its conceptual model. Each data structure depends on the problem's context and design, and the expected efficiency of our algorithm. In conclusion, choosing the right data structure impacts directly on the outcome of any software development project.

In Python, we could create a simple data structure by using an empty object without methods and add the attributes along with our program. However, using empty classes is not recommended, because:

- *i)* it requires a lot of memory to keep tracked all the potentially new attributes, names, and values.
- *ii)* it decreases the maintainability of the code.
- *iii)* it is an overkill solution.

The example below shows the use of the `pass` sentence to let the class empty, which corresponds to a null operation. We commonly use the `pass` sentence when we expect the method to be defined later. Once we create the object, we can add more attributes.

```
1  # We create an empty class
2  class Video:
3      pass
4
5  vid = Video()
```

```
6
7  # We add new attributes
8  vid.ext = 'avi'
9  vid.size = '1024'
10
11 print(vid.ext, vid.size)
```

```
avi 1024
```

We can also create a class only with few attributes, but still without methods. Python allows us to add new attributes to our class on the fly.

```
1  # We create a class with some attributes
2  class Image:
3
4      def __init__(self):
5          self.ext = ''
6          self.size = ''
7          self.data = ''
8
9
10 # Create an instance of the Image class
11 img = Image()
12 img.ext = 'bmp'
13 img.size = '8'
14 img.data = [255, 255, 255, 200, 34, 35]
15
16 # We add this new attribute dynamically
17 img.ids = 20
18
19 print(img.ext, img.size, img.data, img.ids)
```

```
bmp 8 [255, 255, 255, 200, 34, 35] 20
```

Fortunately, Python has many built-in data structures that let us manage data efficiently, such as: `list`, `tuples`, `dictionaries`, `sets`, `stacks`, and `queues`.

2.1 Array-Based Data Structures

In this section, we will review a group of data structures based on the sequential order of their elements. These kinds of structures are indexed through `seq[index]`. Python uses an index format that goes from 0 to $n-1$, where n is the number of elements in the sequence. Examples of this type of structures are: `tuple` and `list`.

Tuples

Tuples are useful for handling ordered data. We can get a particular element inside the tuple by using its index:

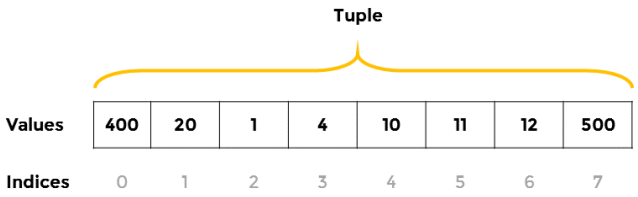


Figure 2.1: Diagram of indexing on tuples. Each cell contains a value of the tuple that could be referenced using its index. In Python, indices go from 0 until $n-1$, where the tuple has length n.

Tuples can handle various kind of data types. We can create a tuple using the tuple constructor as follows: `tuple(`$element_0, element_1, \ldots, element_{n-1}$`)`. We can create a empty tuple using `tuple()` without arguments: `a = tuple()`. We can also create a tuple by directly adding the tuple elements:

```
1 b = (0, 1, 2)
2 print(b[0], b[1])
```

```
0 1
```

A tuple can handle various data types. The parentheses are not mandatory during its creation:

```
1 c = 0, 'message'
2 print(c[0], c[1])
```

```
0 message
```

We can also add any object to the tuple:

```
1 teacher = ('Christian', '23112436-0', 2)
2 video = ('data-structures.avi', 1024, 'mp4')
3 entry = (1, teacher, video)
4 print(entry)
```

```
(1, ('Christian', '23112436-0', 2), ('data-structures.avi', 1024, 'mp4'))
```

Tuples are **immutable**, *i.e*, once we create a tuple, it is not possible to add, remove or change elements of the tuple. This immutability allows us to use tuples as a key value in hashing-based data structures, such as dictionaries. In the next example, we create a tuple with three elements: an instance of the class `Image`, a `string`, and a `float`. Then, we attempt to change the element in position 0 by a string. We can see that this attempt raise a `TypeError` exception:

```
1 a = ('this is' , 'a tuple', 'of strings')
2 a[1] = 'new data'
```

```
Traceback (most recent call last):
  File "05_tuple_inmutable.py", line 2, in <module>
    a[1] = 'new data'
TypeError: 'tuple' object does not support item assignment
```

We can map tuples into a set of individual variables. For example, if a function returns a tuple with several values, the tuple can be assigned separately to a set of individual variables. The code below shows an example, the function `compute_geometry()` receives as input the sides a and b of a quadrilateral and returns a set of geometric measures:

```
1  def compute_geometry(a, b):
2      area = a * b
3      perimeter = (2 * a) + (2 * b)
4      mpa = a / 2
5      mpb = b / 2
6
7      return (area, perimeter, mpa, mpb)
8
9  data = compute_geometry(20.0, 10.0)
10 print('1: {0}'.format(data))
11
12 a = data[0]
13 print('2: {0}'.format(a))
14
15 # Here we unpack the values into independent variables contained
16 # in the tuple
17 a, p, mpa, mpb = data
```

```
18 print('3: {0}, {1}, {2}, {3}'.format(a, p, mpa, mpb))
19 a, p, mpa, mpb = compute_geometry(20.0, 10.0)
20 print('4: {0}, {1}, {2}, {3}'.format(a, p, mpa, mpb))
```

```
1: (200.0, 60.0, 10.0, 5.0)
2: 200.0
3: 200.0, 60.0, 10.0, 5.0
4: 200.0, 60.0, 10.0, 5.0
```

We can use slice notation to select a section of the tuple. In this notation, indexes do not correspond directly to the element positions in the sequence, but they work as boundaries to indicate `sequence[start:stop:steps]`. As a default, `steps` $= 1$. Figure 2.2 shows an example.

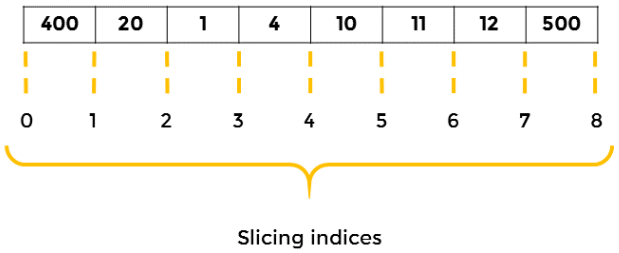


Figure 2.2: Slicing example. Python allows selecting a portion of a tuple or a list using the slice notation. Opposite to a single indexing, slicing start at 0 until n, where n is the length of the sequence.

```
1  data = (400, 20, 1, 4, 10, 11, 12, 500)
2  a = data[1:3]
3  print('1: {0}'.format(a))
4  a = data[3:]
5  print('2: {0}'.format(a))
6  a = data[:5]
7  print('3: {0}'.format(a))
8  a = data[2::2]
9  print('4: {0}'.format(a))
10 #We can revert a sequence:
11 a = data[::-1]
12 print('5: {0}'.format(a))
```

```
1: (20, 1)
2: (4, 10, 11, 12, 500)
3: (400, 20, 1, 4, 10)
4: (1, 10, 12)
5: (500, 12, 11, 10, 4, 1, 20, 400)
```

Named Tuples

Named Tuples let us define a name for each position of the data. They are useful to group elements. First, we require to import the module **namedtuple** from library **collections**. Then, we need to define an object with the tuple attribute names:

```
from collections import namedtuple

# name of tuple type (defined by user) and tuple attributes
Register = namedtuple('Register', 'ID_NUMBER name age')
c1 = Register('13427974-5', 'Christian', 20)
c2 = Register('23066987-2', 'Dante', 5)
print(c1.ID_NUMBER)
print(c2.ID_NUMBER)
```

```
13427974-5
23066987-2
```

Functions can also return Named Tuples:

```
from collections import namedtuple

def compute_geometry(a, b):
    Features = namedtuple('Geometrical', 'area perimeter mpa mpb')
    area = a*b
    perimeter = (2*a) + (2*b)
    mpa = a/2
    mpb = b/2
    return Features(area, perimeter, mpa, mpb)

data = compute_geometry(20.0, 10.0)
print(data.area)
```

```
200.0
```

Lists

This data structure allows us to manage multiple instances of the same type of object, although, they are not limited to combine various type of object classes. Lists are sequential data structures, sorted according to the order we add its elements. Opposite to tuples, lists are **mutable**, *i.e*, their content can dynamically change after their creation.

We must avoid using lists to collect various attributes of an object or using them as vectors in C++, for example as a histogram of words frequency `['python', 20, 'language', 16]`. This way requires an algorithm to access the data inside the list that makes hard use it. In these cases, we must prefer another data structure such as hashing-based data structures, `NamedTuples`, or simply a `dictionary`.

```
# An empty list. We add elements one-by-one
# In this case we add tuples
le = []
le.append((2015, 3, 14))
le.append((2015, 4, 18))
print(le)

# We can also explicitly assign values during creation
l = [1, 'string', 20.5, (23, 45)]
print(l)

# We can retrieve an element using their index
print(l[1])
```

```
[(2015, 3, 14), (2015, 4, 18)]
[1, 'string', 20.5, (23, 45)]
string
```

A useful lists method is `extend()` that allows us to add a complete list to other list already created.

```
# We create a list with 3 elements
songs = ['Addicted to pain', 'Ghost love score', 'As I am']
print(songs)

# Then, we add the list "songs" to the list "new_songs"
```

```
new_songs = ['Elevate', 'Shine']
songs.extend(new_songs)
print(songs)
```

```
['Addicted to pain', 'Ghost love score', 'As I am']
['Addicted to pain', 'Ghost love score', 'As I am', 'Elevate', 'Shine']
```

We can also insert elements at specific positions within the list using the method `insert(position, element)`.

```
# We create a list with 3 elements
songs = ['Addicted to pain', 'Ghost love score', 'As I am']
print(songs)

# Then, we insert a new songs at the position 1
songs.insert(1, 'Sober')
print(songs)
```

```
['Addicted to pain', 'Ghost love score', 'As I am']
['Addicted to pain', 'Sober', 'Ghost love score', 'As I am']
```

In addition, we can ask for an element using the index or retrieve a portion of the list using *slicing* notation. Here we show some examples:

```
# We can take a slice
numbers = [6,7,2,4,10,20,25]
print(numbers[2:6])
```

```
[2, 4, 10, 20]
```

```
# We can pick a portion until the end
print(numbers[2::])
```

```
[2, 4, 10, 20, 25]
```

```
# We can take a slice from the beginning until a specific position
print(numbers[:5])
```

```
[6, 7, 2, 4, 10]
```

```
# We can also change the number of steps
print(numbers[:5:2])
```

```
[6, 2, 10]
```

```
# We can revert a list
print(numbers[::-1])
```

```
[25, 20, 10, 4, 2, 7, 6]
```

Lists can be sorted using the method `sort()`. This method sorts the list in place *i.e*, does not return any value.

```
# We create the list with seven numbers
numbers = [6, 7, 2, 4, 10, 20, 25]
print(numbers)

# Ascendence. Note that variable a do not receive
# any value from assignation.
a = numbers.sort()
print(numbers, a)

# Descendent
numbers.sort(reverse=True)
print(numbers)
```

```
[6, 7, 2, 4, 10, 20, 25]
[2, 4, 6, 7, 10, 20, 25] None
[25, 20, 10, 7, 6, 4, 2]
```

Lists are optimized to be flexible and easy to manage. They are easy to use within `for` loops. Note that we avoid using `id` as a variable because it is a reserved word in Python language.

```
class Piece:
    # Avoid using id as variable because it is a reserved word
    pid = 0

    def __init__(self, piece):
        Piece.pid += 1
```

```
7           self.pid = Piece.pid
8           self.type = piece
9
10  pieces = []
11  pieces.append(Piece('Bishop'))
12  pieces.append(Piece('Pawn'))
13  pieces.append(Piece('King'))
14  pieces.append(Piece('Queen'))
15
16  for piece in pieces:
17      print('pid: {0} - types of piece: {1}'.format(piece.pid, piece.type))
```

```
pid: 1 - types of piece: Bishop
pid: 2 - types of piece: Pawn
pid: 3 - types of piece: King
pid: 4 - types of piece: Queen
```

Stacks

Stacks are a data structures that manage the elements using the **Last-in First-out** (LIFO) principle. When we add elements, they are located on top of the stack. When we remove elements from it, we take the most recently added element. The Figure 2.3 shows an analogy between stacks and a pile of clean dishes. The last added dish will be the first dish to be used.

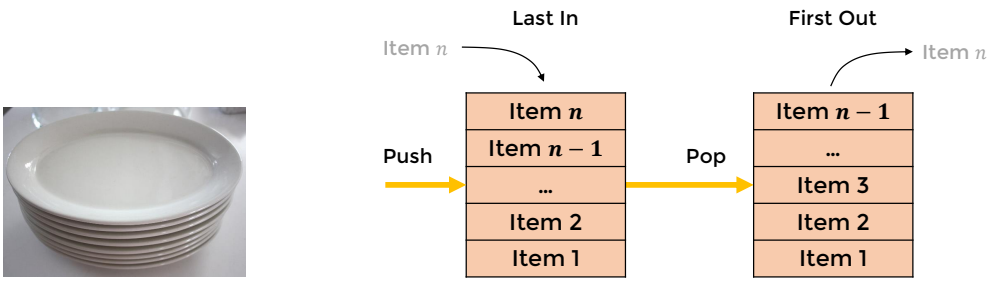


Figure 2.3: Here we show the analogy between `stacks` and a pile of dishes. The `push()` method add an element to the top of the pile. The `pop()` method let us to get the last element added to the stack.

Stacks have two main methods: `push()` and `pop()`. The `push()` method allows us to add an element to the end of the stack and the `pop()` let us to get the top element in the stack. In Python, the stacks are built-in as `Lists`.

There are also more methods, such as: `top()`, `is_empty()`, `len()`. Figure 2.4 includes a brief description and comparison of the other methods included in this data structure.

Basics Methods for Stacks	Python Implementation	Description
Stack.push(item)	List.append(item)	Add sequentialy a new item to the stack
Stack.pop()	List.pop()	Returns and removes the last item added to the stack
Stack.top()	List[-1]	Return the last item added to stack without remove it
len(Stack)	len(List)	Return the total number of items in the stack
Stack.is_empty()	len(List) == 0	Verify whether the stack is empty or not

Figure 2.4: Summary of most used methods of the stack data structure and its equivalence in Python.

Methods described in Figure 2.4 work as follows:

```
1  # Create an empty Stack. In Python Stacks are built-in as Lists.
2  stack = []
3
4  # push() method
5  stack.append(1)
6  stack.append(10)
7  stack.append(12)
8
9  print(stack)
10
11 # pop() method
12 stack.pop()
13 print('pop(): {0}'.format(stack))
14
15 # top() method. Lists does not have a this method implemented directly.
16 #We can have the same behaviour indexing the last element in the Stack.
17 stack.append(25)
18 print('top(): {0}'.format(stack[-1]))
19
20 # len()
21 print('The stack have {0} elements'.format(len(stack)))
22
```

```
23 # is_empty() method. In Python we verify the status of the stack
24 #checking if it has elements.
25 stack  = []
26 if len(stack) == 0:
27     print('The stack is empty :(')
```

```
[1, 10, 12]
pop(): [1, 10]
top(): 25
The stack have 3 elements
The stack is empty :(
```

A practical example of stacks is the back button of web browsers. When we are browsing the internet, each time we visit an URL, the browser add the link to a stack. Then, we can recover the last visited URL when we click on the back button.

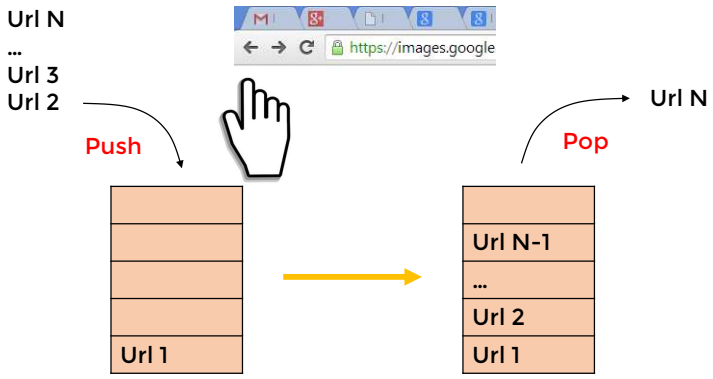


Figure 2.5: An example of using Stacks in a web browser. We can recover the last visited URL every time we press the back button of the browser.

```
1 class Browser:
2
3     def __init__(self, current_url='http://www.google.com'):
4         self.__urls_stack = []
5         self.__current_url = current_url
6
7     def __load_url(self, url):
8         self.__current_url = url
```

```
9           print('loading url: {0}'.format(url))
10
11      def go(self, url):
12          self.__urls_stack.append(self.__current_url)
13          print('go ->', end=' ')
14          self.__load_url(url)
15
16      def back(self):
17          last_url = self.__urls_stack.pop()
18          print('back->', end=' ')
19          self.__load_url(last_url)
20
21      def show_current_url(self):
22          print('Current URL: {0}'.format(self.__current_url))
23
24
25  if __name__ == '__main__':
26      chrome = Browser()
27      chrome.go('http://www.uc.cl')
28      chrome.go('http://www.uc.cl/en/courses')
29      chrome.go('http://www.uc.cl/es/doctorado')
30
31      chrome.show_current_url()
32      chrome.back()
33      chrome.show_current_url()
```

```
go -> loading url: http://www.uc.cl
go -> loading url: http://www.uc.cl/en/courses
go -> loading url: http://www.uc.cl/es/doctorado
Current URL: http://www.uc.cl/es/doctorado
back-> loading url: http://www.uc.cl/en/courses
Current URL: http://www.uc.cl/en/courses
```

Another practical example of stacks is sequence reversion. We show a simple implementation of this task in the next example:

```
1  class Text:
```

```
2       def __init__(self):
3           self.stack = []
4
5       def read_file(self, filename):
6           print('input:')
7
8           with open(filename) as fid:
9               for line in fid:
10                  print(line.strip())
11                  self.stack.append(line.strip())
12
13          print()
14          fid.closed
15
16      def reverse_lines(self):
17          print('output:')
18
19          while len(self.stack) > 0:
20              print(self.stack.pop())
21
22
23  if __name__ == '__main__':
24      t = Text()
25      t.read_file('stacks_text.txt')
26      t.reverse_lines()
```

```
input:
he friend who can be silent with us in a moment of despair or confusion,
who can stay with us in an hour of grief and bereavement,
who can tolerate not knowing... not healing, not curing...
that is a friend who cares.

output:
that is a friend who cares.
who can tolerate not knowing... not healing, not curing...
who can stay with us in an hour of grief and bereavement,
```

```
he friend who can be silent with us in a moment of despair or confusion,
```

Queues

This data structure is an abstract data type that lets us collect objects sequentially following the rule **First–in, First–out** (FIFO). When we add elements, we place them at the end of the queue. When we remove elements from it, we take the oldest element in the queue. Various examples fit with the queue model, such as the line in a supermarket or incoming calls in a call center.

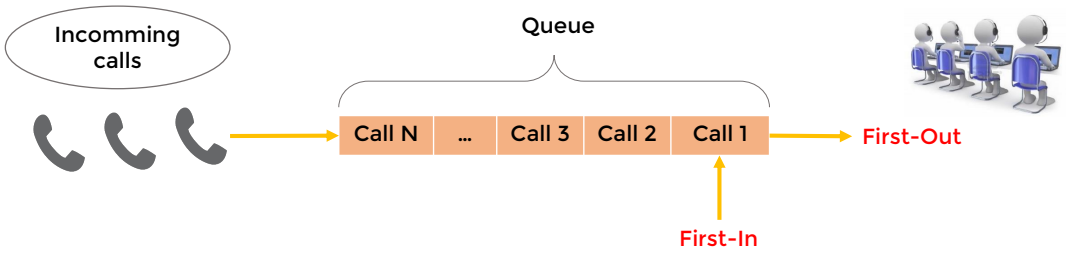


Figure 2.6: A practical example of queues is a call center, where incoming requests wait until an operator can pick the first call added to the queue.

In Python, lists do not work efficiently as queues. Fortunately, the `collections` library includes the `deque` module that is an efficient implementation for data structures based on the FIFO model. This module manages single and double ended queues, while executes all operations efficiently and memory safe. The `deque` module guarantees that the memory access from both sides of the queue is $O(1)$. Although lists support similar operations and methods like queues, they are optimized to perform operations in fixed sized sequences. For example, operations that change the length or position of the element in the sequences, such as `pop(0)` or `insert(0,v)`, involve a cost to update de memory registers of $O(n)$. Queues have two primary methods: `enqueue()` that allows us to add elements to the queue; and `dequeue()`, that returns and removes the first element in the queue. Figure 2.7 includes a brief summary of other methods and operations of this data structure.

```
1 # The collection library includes the deque module that manages single queues
2 # or double ended queues efficiently
3 from collections import deque
4
5 # An empty queue
6 q = deque()
7
```

Basic Methods in Queues	Python Implementation	Description
Queue.enqueue(item)	deque.append(item)	Add an ítem to the queue
Queue.dequeue()	deque.popleft()	Returns and remove the first item in the queue
Queue.first()	deque[1]	Returns the first ítem in the queue without remove it
len(Queue)	len(deque)	Returns the total number of elements in the queue
Queue.is_empty()	len(deque) == 0	Verify whether the queue is empty or not

Figure 2.7: Summary of used methods and operations in queues.

```
8  # We add some elements to the queue using append method, similar to lists.
9  q.append('orange')
10 q.append('apple')
11 q.append('pear')
12
13 # Print the queue
14 print(q)
15
16 # We extract the first element and print again the queue
17 print('Remove the item: {}'.format(q.popleft()))
18 print(q)
19
20 # Add a new element to the queue
21 q.append('strawberry')
22
23 # Get the first element
24 print('The first item is {}'.format(q[0]))
25
26 # len()
27 print('The queue have {0} elements'.format(len(q)))
28
29 # is_empty(). We first remove all the items in the queue using clear() method
30 # and the we check the number of elements.
31 q.clear()
32 if len(q) == 0:
33     print('The queue is empty')
```

```
deque(['orange', 'apple', 'pear'])
Remove the item: orange
deque(['apple', 'pear'])
The first item is pear
The queue have 3 elements
The queue is empty
```

The code below shows a practical example of this data structure applied to vehicle inspection garages:

```
from collections import deque
from random import choice, randrange


class Vehicle:

    # This class models the vehicles that upcoming to the plant and the
    # average time spent during the test
    tp = {'motorcycle': 10, 'car': 25, 'suv': 30}

    def __init__(self):
        self.vehicle_type = choice(list(Vehicle.tp))
        self._testing_time = Vehicle.tp[self.vehicle_type]

    @property
    def testing_time(self):
        return self._testing_time

    @testing_time.setter
    def testing_time(self, new_time):
        self._testing_time = new_time

    def show_type(self):
        print('Testing: {0}'.format(self.vehicle_type))


class Plant:
```

```
28
29      # This class models the test plant
30
31      def __init__(self, vehicles_per_hour):
32          self.test_ratio = vehicles_per_hour
33          self.current_task = None
34          self.testing_time = 0
35
36      def busy(self):
37          return self.current_task != None
38
39      def next_vehicle(self, vehicle):
40          self.current_task = vehicle
41          self.testing_time = self.current_task.testing_time
42          self.current_task.show_type()
43
44      def tick(self):
45          if self.current_task != None:
46              self.testing_time = self.testing_time - 1
47              if self.testing_time <= 0:
48                  self.current_task = None
49
50
51  def arrive_new_car():
52      # This function manage randomly if arrives a new vehicle
53      num = randrange(1, 201)
54      return num == 200
55
56
57  def testing():
58
59      # This function manage the testing process
60
61      # We createa a Plant with capacity of 5 vehicles/hour
62      plant = Plant(5)
```

```
63
64      # Then, we create an empty queue
65      q = deque()
66
67      # Waiting time
68      time_list = []
69
70      # We model arriving vehicles randomly
71      for instant in range(1000):
72
73          if arrive_new_car():
74              v = Vehicle()
75              q.append(v)
76
77          if (not plant.busy()) and (len(q) > 0):
78              # We get the next vehicle in the queue
79              next_vehicle = q.popleft()
80              time_list.append(next_vehicle.testing_time)
81              plant.next_vehicle(next_vehicle)
82
83          # We make time pass by one tick
84          plant.tick()
85
86      average_time = sum(time_list) / len(time_list)
87      print(
88      'Average waiting time: {0:6.2f} min, {1} vehicles queued.'.format(
89          average_time,
90          len(time_list))
91          )
92
93
94  if __name__ == '__main__':
95      testing()
```

```
Testing: suv
Testing: motorcycle
```

```
Testing: motorcycle
Testing: motorcycle
Testing: suv
Testing: motorcycle
Testing: suv
Average waiting time:  18.57 min, 7 vehicles queued.
```

Double Ended Queue (Deque)

The double ended queue data structure is the general case of stack and queue, which allows us to add and remove items from both sides of the structure. This flexibility is useful for modeling real problems where the entities have a different frequency for arrival and attention. This difference originates that some objects leave the queue early. A real example is a call center that receives incoming calls with different priorities and others calls finish abruptly.

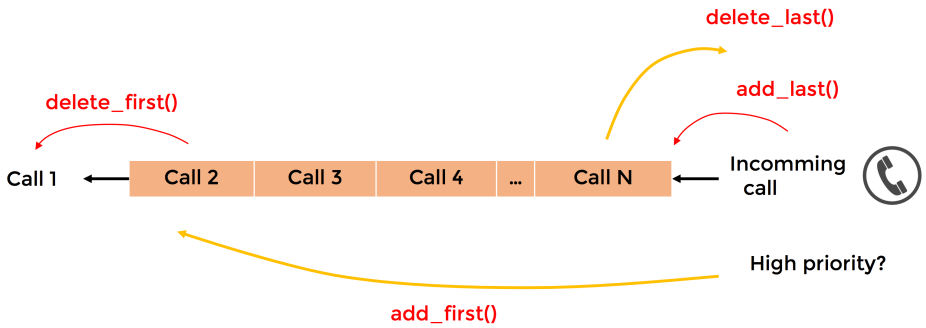


Figure 2.8: Example of using a double ended queues.

Similar to queues, we create a *deque* structure using the **deque** module included in the **collections** library. The table in Figure 2.9 shows a summary of the basic methods and operations of a *deque*.

```
1  from collections import deque
2
3  # We create an empty deque and item by item
4  d = deque()
5  d.append('r')
6  d.append('a')
7  d.append('d')
8  d.append('a')
9  d.append('r')
10 d.append('e')
```

Collection.deque	Python Implementation	Description
Deque.add_first(item)	Deque.appendleft(item)	Add an item at the begining of the deque
Deque.add_last(item)	Deque.append(item)	Add an item at the end of the deque
Deque.delete_first()	Deque.popleft()	Returns and removes the first ítem
Deque.delete_last()	Deque.pop()	Returns and remove the last item
Deque.first()	Deque[0]	Returns the first item without extract it
Deque.last()	Deque[-1]	Returns the last item without extract it
len(Deque)	len(Deque)	Returns the total number of ítems in the deque
Deque.is_empty()	len(Deque) == 0	Verify whether the deque is empty or not
	Deque[j]	Access to the item j
	Deque[j] = value	Modify the item j
	Deque.clear()	Delete all the ítems
	Deque.rotate(k)	K step circular scrolling
	Deque.remove(e)	Remove the first item that match to e
	Deque.count(e)	Counts the number of items that match to e

Figure 2.9: Brief summary of the basic `deque` methods in python.

```
11 d.append('s')
12
13 print(d)
14 print(len(d))

   deque(['r', 'a', 'd', 'a', 'r', 'e', 's'])
   7

1  # Next, we check the first and the last items
2  print(d[0], d[-1])

   r s

1  # Then, we rotate the deque in k=3 step
2  d.rotate(3)
3  print(d)

   deque(['r', 'e', 's', 'r', 'a', 'd', 'a'])

1  # Finally, we extract the first and the last items
2  first = d.popleft()
3  last = d.pop()
4
```

```
print(first, last)
print(d)
```

```
r a
deque(['e', 's', 'r', 'a', 'd'])
```

Next, a simple example of the use of a *deque* for palindrome detection. The word is saved in a *deque*, while we iteratively remove and compare the first and the last characters.

```
from collections import deque


class Word:

    def __init__(self, word=None):
        self.word = word
        self.characters = deque(self.word)

    def is_palindrome(self):
        if len(self.characters) > 1:
            return self.characters.popleft() == self.characters.pop() \
             and Word(''.join(self.characters)).is_palindrome()
        else:
            return True

p1 = Word("radar")
p2 = Word("level")
p3 = Word("structure")

print(p1.is_palindrome())
print(p2.is_palindrome())
print(p3.is_palindrome())
```

```
True
True
False
```

Note: In Python, if we want to check whether a word is a palindrome or not, we only need to compare word == word[::-1].

Dictionaries

A **dictionary** is a data structure based on **key–value** associations. This relation allows us to use the **key** to efficiently search an item because it points to the memory address where its associated **value** is located.

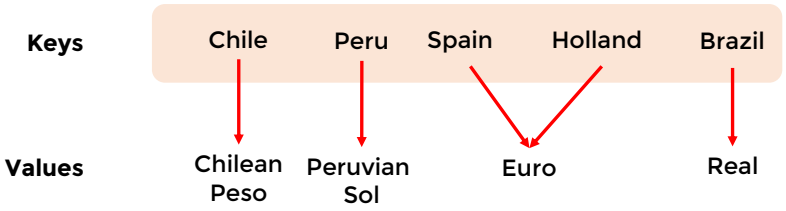


Figure 2.10: Example of a dictionary. Every key is unique in the dictionary.

In Python, an empty dictionary is created by braces `{}` or `dict()`. We must specify the key and value using colon :, *i.e*, `{key1: value1, key2: value2, ...}`. The key has to be an inmutable object: `int`, `str`, `tuple`, etc. We can access the value using the key within **brackets**. The next example shows how we create a dictionary:

```
1  dogs = {'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug'}
2  telephones = {23545344: 'John', 23545340: 'Trinity', 23545342: 'Taylor'}
3  tuples = {('23545344', 0): 'office', ('2353445340', 1): 'admin'}
4
5  print(dogs)
6  print(tuples)
7
8  # We access directly to the value using its key
9  print(dogs['bc'])
10 print(telephones[23545344])
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug'}
{('2353445340', 1): 'admin', ('23545344', 0): 'office'}
border-collie
John
```

In the next example, we can see that *dictionaries* are not a sorted sequence like *tuples* or *list*; and that the key can even be of various types within the same *dictionary*.

```
d = {1: 'first key', '2': 'second key', 23.0: 'third key',
     (23, 5): 'fourth key'}
print(d)
```

```
{1: 'first key', (23, 5): 'fourth key', '2': 'second key',
23.0: 'third key'}
```

Dictionaries are **mutable** data structures, *i.e*, its content can change through the execution of a program. There are two behaviors when we assign a value to key: 1) if the key does not exist, Python creates the key in the dictionary and assigns the value; and 2) if the key already exists, Python assigns the new value.

```
# If a key does not exist, Python creates a new one and assigns the value
dogs['te'] = 'terrier'
print(dogs)

# If the key already exist, Python just assigns the value
dogs['pg'] = 'pug-pug'
print(dogs)
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'te': 'terrier',
'pg': 'pug'}
{'bc': 'border-collie', 'lr': 'labrador retriever', 'te': 'terrier',
'pg': 'pug'}
```

We can also remove items directly from a *dictionary* by using the `del` sentence. For example:

```
# We remove items using del
del dogs['te']
print(dogs)
```

```
{'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug-pug'}
```

We can check whether a given key exists in a *dictionary* by using the `in` statement. By default, every time we use this sentence directly with the name of the *dictionary*, Python assumes that we refer to its list of keys. The result of using `in` is `True` if the required key exists in the *dictionary* keys, and `False` otherwise:

```
# The sentence in checks whether exist a specific key in the dictionary
print('pg' in dogs)
print('te' in dogs)
```

```
True
False
```

Similarly, dictionaries have the `get()` method that allows us to check whether a key exist or not. This method requires two parameters: the key and a default value for missing keys. We can use any Python object as a default value.

```
# The method get() checks whether a key exists or not.

print(dogs.get('pg', False))  # logical value as default
print(dogs.get('te', 2))  # int value as default
print(dogs.get('te', 'The dog does not exist'))  # String value as default
```

```
True
False
pug-pug
```

This verification is useful when we need to build a *dictionary* during the execution of our program, such as we show in the following example:

```
# A string to be verified and a empty dictionary to count vowels
msg = 'supercalifragilisticexpialidocious'
vowels = dict()

for v in msg:
    if v not in 'aeiou':  # check wheter v is vowel or not
        continue

    # If v exist, add a key named as v initialized at 0
    if v not in vowels:
        vowels[v] = 0

    # If v alread exist, increase the counter
    vowels[v] += 1

```

```
16 print(vowels)
```

```
{'i': 7, 'a': 3, 'o': 2, 'e': 2, 'u': 2}
```

There are three useful methods that let us to retrieve the dictionary contents at different levels. These methods are: `keys()`, `values()`, and `items()`. The example below shows the results of using these methods and their outputs:

```
1 currency = {'Chile': 'Peso', 'Brazil': 'Real',
2             'Peru': 'Sol', 'Spain': 'Euro', 'Italy': 'Euro'}
3
4 print(currency.keys())  # returns a list with the keys
5 print(currency.values())  # returns a list with the values
6 print(currency.items())  # returns a list of tuples key-value
```

```
dict_keys(['Chile', 'Spain', 'Italy', 'Peru', 'Brazil'])
dict_values(['Peso', 'Euro', 'Euro', 'Sol', 'Real'])
dict_items([('Chile', 'Peso'), ('Spain', 'Euro'), ('Italy', 'Euro'),
            ('Peru', 'Sol'), ('Brazil', 'Real')])
```

The methods `keys()`, `values()`, and `items()` described above are completely suitable during iteration over dictionaries. By default, when we use a `for` loop to iterate over a *dictionary* Python iterates directly over the keys. In each iteration the local variable takes a key value in the list.

```
1 # Iteration over a dictionary
2
3 # By default Python iterates directly over the keys
4 print('This dictionary has the following keys:')
5
6 for k in currency:
7     print('{0}'.format(k))
```

```
This dictionary has the following keys:
Chile
Spain
Italy
Peru
Brazil
```

Or we can also use the method `keys()` explicitly

```
# Although we can also use the method keys()
print('This dictionary has the following keys:')

for k in currency.keys():
    print('{0}'.format(k))
```

```
This dictionary has the following keys:
Chile
Spain
Italy
Peru
Brazil
```

The method `values()` allows us to iterate over the list of values associated to each key.

```
# We use the method values() when we want to iterates using the values
print('The values in the dictionary:')
for v in currency.values():
    print('{0}'.format(v))
```

```
The values in the dictionary:
Peso
Euro
Euro
Sol
Real
```

Finally, the method `items()` provide us with a way to iterate using the pair key-value.

```
# The method items() allows us to retrieve a tuple (key, value)
print('The pairs key-value:')

for k, v in currency.items():
    print('the currency in {0} is {1}'.format(k, cv))
```

```
The pairs key-value:
the currency in Chile is Peso
the currency in Spain is Euro
```

```
the currency in Italy is Euro
the currency in Peru is Sol
the currency in Brazil is Real
```

Defaultdicts

Python provide us with a special case of *dictionary* called `defaultdicts` that allows us to assign a default value to the nonexistent keys. This type of *dictionary* is part of the `collections` library and let us to save time writing line codes to manage the cases when our code tries to access a nonexistent keys. The `defaultdics` accept also a function as default value, which can receive an action and return any object as key in the *dictionary*. For example, suppose we want to create a *dictionary* where each new key has as value a list with a string indicating the current number of items in the *dictionary*.

```
1  from collections import defaultdict
2
3  num_items = 0
4
5  def my_function():
6      global num_items
7      num_items += 1
8      return ([str(num_items)])
9
10 d = defaultdict(my_function)
11
12 print(d['a'])
13 print(d['b'])
14 print(d['c'])
15 print(d['d'])
16 print(d)
```

```
['1']
['2']
['3']
['4']
defaultdict(
  <function my_function at 0x000000000295CBF8>,
```

```
{'d': ['4'], 'b': ['2'], 'c': ['3'], 'a': ['1']})
```

Sets

A *set* is a data structure that contains an unordered collection of unique and immutable objects. For example: imagine that we have a list that contains a collection of tuples of items `(song, artist)` where different songs may be associated with the same artist, and we would like to build a list of all unique artists in our library. To do so, we may create a new list and for each item in the collection check if we already added the artist to the new list. That iteration is inefficient. *Sets* provide us with a way to do this task easily because the data structures make sure that each item in the *set* is unique even if we add the same item again.

Python requires that the *sets* contain **hashable** objects, *i.e*, an immutable object that has a hash value registered in the `__hash__()` method. This values never changes during its lifetime and can be compared to other objects. Hashable objects have the advantage that they can be used as keys in dictionaries.

```
songs_list = [("Uptown Funk", "Mark Ronson"),
              ("Thinking Out Loud", "Ed Sheeran"),
              ("Sugar", "Maroon 5"),
              ("Patterns In The Ivy", "Opeth"),
              ("Take Me To Church", "Hozier"),
              ("Style", "Taylor Swift"),
              ("Love Me Like You Do", "Ellie Goulding")]

artists = set()

for song, artist in songs_list:
    # The add() method append a new item to the set. We do not require to
    # check whether the item exist or not previously in the set.
    artists.add(artist)

print(artists)
```

```
{'Mark Ronson', 'Opeth', 'Hozier', 'Ed Sheeran', 'Maroon 5', 'Taylor Swift',
  'Ellie Goulding'}
```

We also can build a *set* using brackets, where a coma must separate the items. An empty *set* has to be created with the `set()` statement, otherwise a dictionary will be created.

```
# We can create a set using brackets including items separated by coma.
songs = {'Style', 'Uptown Funk', 'Take Me To Church', 'Sugar',
         'Thinking Out Loud', 'Patterns In The Ivy', 'Love Me Like You Do'}

print(songs)
print('Sugar' in songs)

for artist in artists:
    print("{} plays excellent music".format(artist))
```

```
{'Patterns In The Ivy', 'Take Me To Church', 'Sugar', 'Love Me Like You Do',
 'Style', 'Uptown Funk', 'Thinking Out Loud'}
True
Mark Ronson plays excellent music
Opeth plays excellent music
Hozier plays excellent music
Ed Sheeran plays excellent music
Maroon 5 plays excellent music
Taylor Swift plays excellent music
Ellie Goulding plays excellent music
```

Note that the results of the previous example show that the items in the *set* are unordered (similar to dictionaries). *Sets* cannot be indexed to retrieve their items. We can build an ordered list from a *set* as follows:

```
# Build a list from a set
artists_list = list(artists)
artists_list.sort()
print(artists_list)
```

```
['Ed Sheeran', 'Ellie Goulding', 'Hozier', 'Mark Ronson', 'Maroon 5',
'Opeth', 'Taylor Swift']
```

Sets data structures behave as mathematical *sets* and provide us with the same mathematical operations.

```
# Mathematical Operations
my_artists = {
    'Hozier', 'Opeth', 'Ellie Goulding', 'Mark Ronson', 'Taylor Swift'
}
```

```

artists_album = {'Maroon 5', 'Taylor Swift', 'Amy Wadge'}

print("All: {}".format(my_artists.union(artists_album)))
print("both: {}".format(artists_album.intersection(my_artists)))
```

```
All: {'Mark Ronson', 'Opeth', 'Hozier', 'Taylor Swift', 'Maroon 5',
'Amy Wadge', 'Ellie Goulding'}
both: {'Taylor Swift'}
```

```
# The A.difference(B) returns a set of items that exist only in A but
# not in B.
print("Only in A: {}".format(my_artists.difference(artists_album)))
```

```
Only in A: {'Ellie Goulding', 'Mark Ronson', 'Hozier', 'Opeth'}
```

```
# The symmetric difference returns a set of items that exist only in
# one of the sets, but not both.
print("Any but not both: {}".format(my_artists.symmetric_difference(
    artists_album)))
```

```
Any but not both: {'Mark Ronson', 'Opeth', 'Hozier', 'Maroon 5',
'Amy Wadge', 'Ellie Goulding'}
```

```
# Operation Order
bands = {"Opeth", "Guns N' Roses"}

print("my_artist is to bands:")
print("issuperset: {}".format(my_artists.issuperset(bands)))
print("issubset: {}".format(my_artists.issubset(bands)))
print("difference: {}".format(my_artists.difference(bands)))

print("-" * 20)

print("bands is to my_artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print("difference: {}".format(bands.difference(my_artists)))
```

```
my_artist is to bands:
issuperset: False
issubset: False
difference: {'Mark Ronson', 'Taylor Swift', 'Hozier', 'Ellie Goulding'}
--------------------
bands is to my_artists:
issuperset: False
issubset: False
difference: {"Guns N' Roses"}
```

In some methods, the arguments' order does not matter, for example, `my_artists.union(artists_album)` returns the same result that `my_artist_album.union(my_artist)`. There are other methods where the arguments' order does matter, for example, `issubset()` and `issuperset()`.

2.2 Node-based Data Structures

In this section, we describe a set of data structures based on a single and basic structure called **node**. A **node** allocates an item and its elements and maintains one or more reference to neighboring nodes to represent more complex data structures collectively. One relevant aspect of these complex structure is the way on how we walk through each node. The **traversal** is the way to visit all the nodes in a node-base structure systematically. The following sections show how to build and traverse two essentials node-based structures: *linked lists* and, *trees*.

Singly Linked List

This data structure is one of the primary node-based structure. In a *linked list*, a collection of nodes forms a linear sequence where each node has a unique precedent and subsequent nodes. The first node is called `head` and the last node is called `tail`. In this structure, nodes have references to their `value` and to the `next` element in the sequence. Figure 2.11 shows a diagram of a linked list. In the `tail` node there is no reference to the `next` object.

The way to traverse a linked list is node-by-node recursively. Every time we get a node we have to pick the next one, indicated with the `next` statement. The traverse stops when there are no more nodes in the sequence. The code below shows how to build a linked list. Lines 21 to 31 show how to traverse the structure.

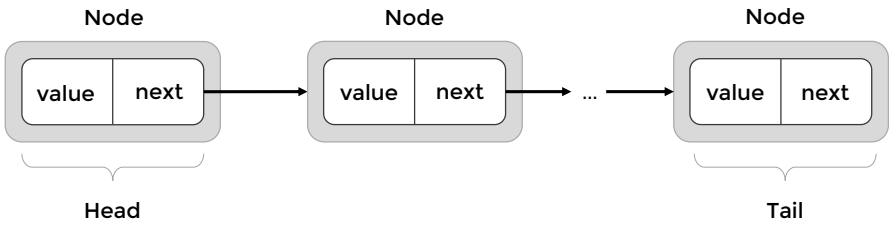


Figure 2.11: The simpler implementation of a linked list consists into a node that has two attributes: the value of the node and a reference to the next node. We can put as many nodes as we require.

```
class Node:
    # This class models the basic structure, the node.
    def __init__(self, value=None):
        self.next = None
        self.value = value

class LinkedList:
    # This class implement a singly linked list
    def __init__(self):
        self.tail = None
        self.head = None

    def add_node(self, value):
        if not self.head:
            self.head = Node(value)
            self.tail = self.head
        else:
            self.tail.next = Node(value)
            self.tail = self.tail.next

    def __repr__(self):
        rep = ''
        current_node = self.head

        while current_node:
            rep += '{0}'.format(current_node.value)
            current_node = current_node.next
            if current_node:
                rep += ' -> '

        return rep

if __name__ == '__main__':
    l = LinkedList()
    l.add_node(2)
```

```
36      l.add_node(4)
37      l.add_node(7)
38
39      print(l)
```

```
2 -> 4 -> 7
```

Trees

Trees are one of the most important data structure in computer science. A **tree** is a collection of nodes structured **hierarchically**. Opposite to the array-based structures (*e.g.* stacks and queues), the nodes that represent the items lay ordered **above** and **below** according to the **parent-child** hierarchy. A tree has a top node called **root** that is the only node that does not have a parent. Nodes other than the **root** have a single parent and one o more children. Children nodes descending from the same parent are called **siblings**.

We say that a node a is an **ancestor** of node b if a is in the path from b to the root. Nodes that have no children are called **leaf** nodes (sometimes called **external**). Nodes that are not the root or leaves are called **internal** nodes. Recursively, every node can be the root of its subtree. Figure 2.12 shows a tree representation of the animal kingdom. The root node has two children: Vertebrates and Invertebrates. The Invertebrates node has three children that are siblings each other: mollusks, arthropod, and worms. The node *Annelids* is a leave node. *Vertebrates* can be a root node of the subtree formed by its children.

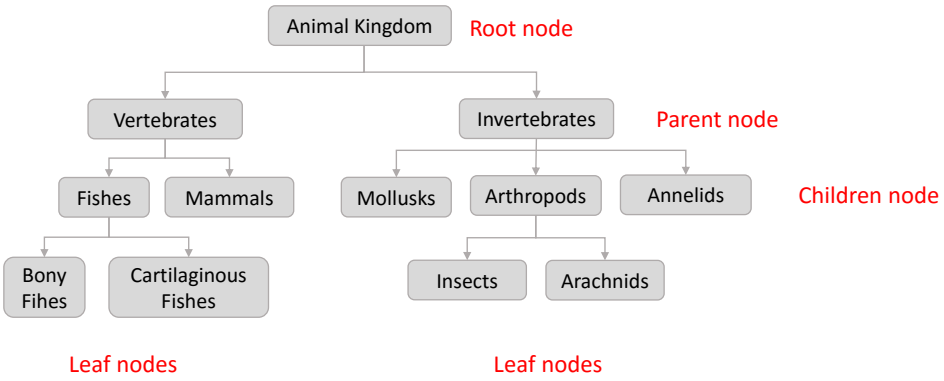


Figure 2.12: An example of a tree structure to represent the Animal Kingdom shows a tree representation of the animal kingdom. According to the definition, the root node has no a parent node. All the internal nodes have a parents-child relationship. The leaf nodes have no children. Every node can be the root of its own subtree, *e.g*, Fishes.

An **edge** connects a pair of nodes *(u, v)* that have a parent-child relationship. Each node has a unique incoming edge (parent) and zero or various outgoing edges (children). An ordered sequence of consecutive nodes joint by a set of

edges from a starting node to a destination node through the tree form a **path**. In the same Figure 2.12, there are two edges that connect the node *Fishes* with its children *Bony* and *Cartilaginous*. The set of edges from *Bony* to *Animals* form the path *Animals-Vertebrates-Fishes-Bony*.

The **depth** of a node *b* is the number of levels or ancestors that exist between *b* and the root node. The **height** of a tree is the number of levels in the tree, or the maximum depth reached among the leaf nodes. As shown in Figure 2.12, the *Fishes* node has depth 2, and the height of the tree is 3.

Binary Tree

Binary trees are among the most used tree structures in computer science. In a binary tree, each node has a maximum number of two children; each child node has a label: **left- child** and **right-child**, and regarding precedence, the left child precedes the right child.

In binary trees, the number of nodes grows exponentially with depth. Let d be the **level** of an binary tree **T** defined as the set of nodes located at the same depth d. At the level $d = 0$ there is at least only one node (the root). The level $d = 1$ has at least two nodes, and so on. At any level d, the tree has a maximum number of 2^d levels. The special case when every node has two children is know as a **complete tree**.

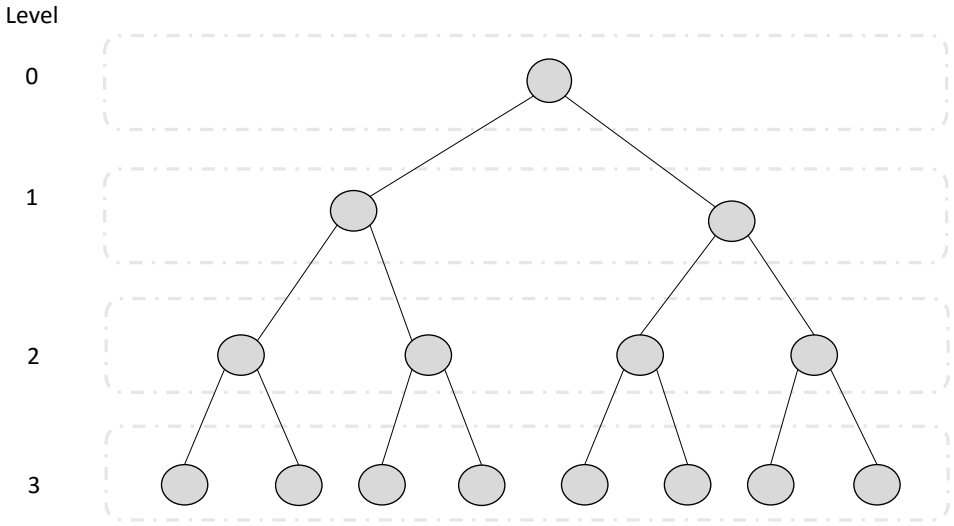


Figure 2.13: An example of a binary tree. As we can see, the node 0 is the root of the tree. For this example, we adopt the convention of setting the numbers while traversing the tree in amplitude.

A practical example of binary trees is **decision trees**. In this kind of trees, each interior node and the root represent a query, and their outgoing edges represent the possible answers. Another example is **expression trees**; they represent arithmetic operations where variables correspond to leaf nodes and operators to interior nodes.

Linked Structure Based Binary Tree

The linked structure based binary tree correspond to the recursive version for binary trees. Each node of the tree is an object where each attribute is a reference to the parent node, children nodes, and its value. We use `None` to indicate that an attribute does not exist. For example, if we write the root node, the attribute `parent` is equal to `None`. Now we show the implementation of a binary tree using a linked structure:

```
class Node:

    def __init__(self, value, parent=None):
        self.value = value
        self.parent = parent
        self.left_child = None
        self.right_child = None

    def __repr__(self):
        return 'parent: {0}, value: {1}'.format(self.parent, self.value)


class BinaryTree:

    def __init__(self, root_node=None):
        self.root_node = root_node

    def add_node(self, value):
        if self.root_node is None:
            self.root_node = Node(value)
        else:
            temp = self.root_node
            added = False

            while not added:
                if value <= temp.value:
                    if temp.left_child is None:
                        temp.left_child = Node(value, temp.value)
                        added = True
```

```
30
31                      else:
32                          temp = temp.left_child
33
34                  else:
35                      if temp.right_child is None:
36                          temp.right_child = Node(value, temp.value)
37                          added = True
38
39                      else:
40                          temp = temp.right_child
41
42      def __repr__(self):
43          def traverse_tree(node, side="root"):
44              ret = ''
45
46              if Node is not None:
47                  ret += '{0} -> {1}\n'.format(node, side)
48                  ret += traverse_tree(node.left_child, 'left')
49                  ret += traverse_tree(node.right_child, 'right')
50
51              return ret
52
53          return traverse_tree(self.root_node)
54
55
56  T = BinaryTree()
57  T.add_node(4)
58  T.add_node(1)
59  T.add_node(5)
60  T.add_node(3)
61  T.add_node(20)
62
63  print(T)
```

```
parent: None, value: 4 -> root
```

```
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

Binary Tree Traversal

In the following sections, we describe the three basic methods to traverse a binary tree: pre-order traversal, in-order traversal, post-order traversal.

Pre-Order Traversal

In this method we first visit the root node and then its children recursively:

```
# 31_binary_trees_pre_order_traversal.py

class BinaryTreePreOrder(BinaryTree):
    # We inherited the original class of our binary tree, and override the
    # __repr__ method to traverse the tree using pre-order traversal.

    def __repr__(self):
        def traverse_tree(node, side="root"):
            ret = ''

            if node is not None:
                ret += '{0} -> {1}\n'.format(node, side)
                ret += traverse_tree(node.left_child, 'left')
                ret += traverse_tree(node.right_child, 'right')

            return ret

        return traverse_tree(self.root_node)


if __name__ == '__main__':
    # We add some nodes to the tree
    T = BinaryTreePreOrder()
```

```
24      T.add_node(4)
25      T.add_node(1)
26      T.add_node(5)
27      T.add_node(3)
28      T.add_node(20)
29
30      print(T)
```

```
parent: None, value: 4 -> root
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

In-Order Traversal

In this method we first visit the **left-child**, then the **root** and finally the **right-child** recursively:

```
1  # 32_binary_trees_in_order_traversal.py
2
3  class BinaryTreeInOrder(BinaryTree):
4      # We inherited the original class of our binary tree, and override the
5      # __repr__ method to traverse the tree using pre-order traversal.
6
7      def __repr__(self):
8          def traverse_tree(node, side="root"):
9              ret = ''
10
11             if node is not None:
12                 ret += traverse_tree(node.left_child, 'left')
13                 ret += '{0} -> {1}\n'.format(node, side)
14                 ret += traverse_tree(node.right_child, 'right')
15
16             return ret
17
18         return traverse_tree(self.root_node)
19
```

```
20
21 if __name__ == '__main__':
22     # We add some nodes to the tree
23     T = BinaryTreeInOrder()
24     T.add_node(4)
25     T.add_node(1)
26     T.add_node(5)
27     T.add_node(3)
28     T.add_node(20)
29
30     print(T)
```

```
parent: 4, value: 1 -> left
parent: 1, value: 3 -> right
parent: None, value: 4 -> root
parent: 4, value: 5 -> right
parent: 5, value: 20 -> right
```

Post-Order Traversal

The post-order traversal first finds the sub-trees descending from the children nodes, and finally the root.

```
1  # 33_binary_trees_post_order_traversal.py
2
3  class BinaryTreePostOrder(BinaryTree):
4      # We inherited the original class of our binary tree, and override the
5      # __repr__ method to traverse the tree using pre-order traversal.
6
7      def __repr__(self):
8          def traverse_tree(node, side="root"):
9              ret = ''
10
11             if node is not None:
12                 ret += traverse_tree(node.left_child, 'left')
13                 ret += traverse_tree(node.right_child, 'right')
14                 ret += '{0} -> {1}\n'.format(node, side)
15
```

```
16                 return ret
17
18             return traverse_tree(self.root_node)
19
20
21 if __name__ == '__main__':
22     # We add some nodes to the tree
23     T = BinaryTreePostOrder()
24     T.add_node(4)
25     T.add_node(1)
26     T.add_node(5)
27     T.add_node(3)
28     T.add_node(20)
29
30     print(T)
```

```
parent: 1, value: 3 -> right
parent: 4, value: 1 -> left
parent: 5, value: 20 -> right
parent: 4, value: 5 -> right
parent: None, value: 4 -> root
```

N-ary Trees

The N-ary trees correspond to a generalization of trees. Differently from the binary case, in N-ary trees, each node may have zero or more children.

Linked Structured N-ary Tree

Similar to a binary tree, we can build N-ary trees using a linked structure where each node is a tree itself. Following the tree definition, the complete N-ary tree is a collection of nodes that we append incrementally. Each node includes the following attributes: `node_id`, `parent_id`, `children`, and `value`. Figure 2.14 shows an example of a tree with three levels where each node has a value and an identifier.

The code below shows a recursive implementation of a linked structured tree:

```
1 # 34_linked_trees.py
```

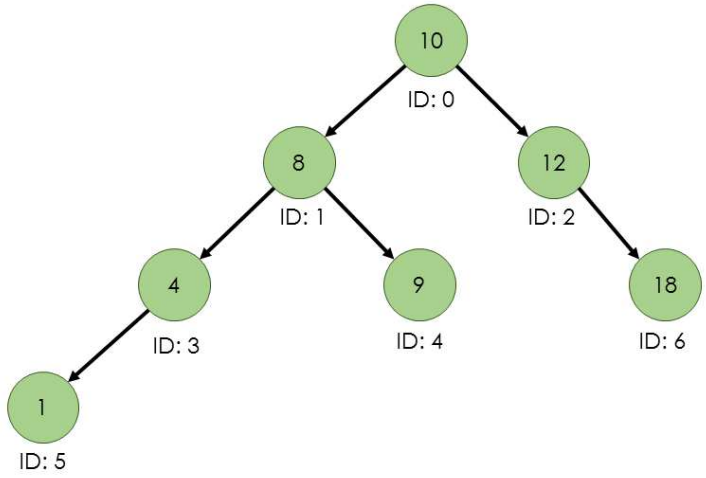


Figure 2.14: An example of a general tree structure. Green circles denote the nodes that include its value. Each node also has an identification number ID. Black arrows represent edges.

```
2
3  class Tree:
4      # We create the basic structure of the tree. Children nodes can be keep in
5      # a different data structure, such as: a lists or a dictionary. In this
6      # example we manage the children nodes in a dictionary.
7
8      def __init__(self, node_id, value=None, parent_id=None):
9          self.node_id = node_id
10         self.parent_id = parent_id
11         self.value = value
12         self.children = {}
13
14     def add_node(self, node_id, value=None, parent_id=None):
15         # Every time that we add a node, we need to verify the parent of the
16         # new node. If it is not the parent, we search recursively through the
17         # the tree until we find the right parent node.
18
19         if self.node_id == parent_id:
20             # If the node is the parent, we update the dictionary with the
21             # children.
22             self.children.update({node_id: Tree(node_id, value, parent_id)})
```

```
23          else:
24              # If the node is not the parent we search recursively
25              for child in self.children.values():
26                  child.add_node(node_id, value, parent_id)
27
28      def get_node(self, node_id):
29          # We recursively get the node as long as it exists in the tree.
30          if self.node_id == node_id:
31              return self
32          else:
33              for child in self.children.values():
34                  node = child.get_node(node_id)
35                  if node:
36                      # if the node exists in the tree, returns the node
37                      return node
38
39      def __repr__(self):
40          # We override this method in order to traverse recursively the node in
41          # the tree.
42          def traverse_tree(root):
43              ret = ''
44              for child in root.children.values():
45                  ret += "id-node: {} -> parent_id: {} -> value: {}\n".format(
46                      child.node_id, child.parent_id, child.value)
47                  ret += traverse_tree(child)
48              return ret
49
50          ret = 'root:\nroot-id: {} -> value: {}\n\nchildren:\n'.format(
51              self.node_id, self.value)
52          ret += traverse_tree(self)
53
54          return ret
55
56
57  if __name__ == '__main__':
```

```
58      T = Tree(0, 10)
59      T.add_node(1, 8, 0)
60      T.add_node(2, 12, 0)
61      T.add_node(3, 4, 1)
62      T.add_node(4, 9, 1)
63      T.add_node(5, 1, 3)
64      T.add_node(6, 18, 2)
```

```
root:
root-id: 0 -> value: 10

children:
id-node: 1 -> parent_id: 0 -> value: 8
id-node: 3 -> parent_id: 1 -> value: 4
id-node: 5 -> parent_id: 3 -> value: 1
id-node: 4 -> parent_id: 1 -> value: 9
id-node: 2 -> parent_id: 0 -> value: 12
id-node: 6 -> parent_id: 2 -> value: 18
```

We use the method `get_node()` to get a specific node. In this implementation, the method returns the object representing the node:

```
1       node = T.get_node(6)
2       print('The ID of the node is {}'.format(node))
3
4       node = T.get_node(1)
5       print('The node has {} children'.format(len(node.children)))
```

```
The ID of the node is root:
root-id: 6 -> value: 18

children:

The node have 2 children
```

N-ary Tree Traversal

There are two basic methods to traverse the tree: pre-order traversal and post-order traversal. These approaches generalize the traverse methods for binary trees. Note that in this case, the in-order traverse is difficult to define because we cannot determine after which child we have to visit the root node.

Pre-Order Traversal

In this method, we first visit the root node and then its children recursively:

```
# 35_trees_pre_order_traversal.py

class TreePreOrder(Tree):
    # We inherited the original class of our linked tree, and override the
    # __repr__ method to traverse the tree using pre-order traversal.

    def __repr__(self):
        def traverse_tree(root):
            ret = ''
            # We first visit the root note
            ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
                root.node_id, root.parent_id, root.value)

            # And finally, we traverse the children recursively
            for child in root.children.values():
                ret += traverse_tree(child)

            return ret

        return traverse_tree(self)


if __name__ == '__main__':
    # We add some nodes to the tree
    T = TreePreOrder(0, 10)
    T.add_node(1, 8, 0)
    T.add_node(2, 12, 0)
```

```
28      T.add_node(3, 4, 1)
29      T.add_node(4, 4, 1)
30      T.add_node(5, 1, 3)
31      T.add_node(6, 18, 2)
32
33      print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 1, parent_id: 0 -> value: 8
node_id: 3, parent_id: 1 -> value: 4
node_id: 5, parent_id: 3 -> value: 1
node_id: 4, parent_id: 1 -> value: 4
node_id: 2, parent_id: 0 -> value: 12
node_id: 6, parent_id: 2 -> value: 18
```

Post-Order Traversal

The post-order traversal first finds the sub-trees descendant from the children nodes, and finally the root:

```
1  class TreePostOrder(Tree):
2      # We inherited the class Tree from te previous example and we override the
3      # __repr__ method using the post-order traversal.
4
5      def __repr__(self):
6          def traverse_tree(root):
7              ret = ''
8
9              # we first recursively traverse the children
10             for child in root.children.values():
11                 ret += traverse_tree(child)
12
13             # Finally, we visit the root note
14             string = "node_id: {}, parent_id: {} -> value: {}\n"
15             ret += string.format(root.node_id, root.parent_id, root.value)
16
17             return ret
18
```

```
19          return traverse_tree(self)
20
21
22  if __name__ == '__main__':
23      # We add instances to the tree
24      T = TreePostOrder(0, 10)
25      T.add_node(1, 8, 0)
26      T.add_node(2, 12, 0)
27      T.add_node(3, 4, 1)
28      T.add_node(4, 4, 1)
29      T.add_node(5, 1, 3)
30      T.add_node(6, 18, 2)
31
32      print(T)
```

```
node_id: 5, parent_id: 3 -> value: 1
node_id: 3, parent_id: 1 -> value: 4
node_id: 4, parent_id: 1 -> value: 4
node_id: 1, parent_id: 0 -> value: 8
node_id: 6, parent_id: 2 -> value: 18
node_id: 2, parent_id: 0 -> value: 12
node_id: 0, parent_id: None -> value: 10
```

Non-Recursive Traversal

There are two non-recursive methods to implement the tree traversal: **Breadth-First Search** (**BFS**) and **Depth-First Search** (**DFS**). These non-recursive algorithms use auxiliary data structures to keep a record of the nodes we must visit, to avoid infinite loops while traversing trees or other complex node-based structures.

Breadth-First Search

In the **BFS** strategy, the algorithm traverses the nodes level by level hierarchically, *i.e*, the root node first, then the set of nodes on the second level, etc. This algorithm uses a queue to keep the nodes that it has to visit in the following iterations.

```
1
2  class TreeBFS(Tree):
```

```
3       # We inherited the original class Tree from the previous example and
4       # override the __repr__ method to apply the BFS algorithm.
5
6       def __repr__(self):
7           def traverse_tree(root):
8               ret = ''
9               Q = deque()
10              Q.append(root)
11
12              # We use a list to keep the visited nodes
13              visited = []
14
15              while len(Q) > 0:
16                  p = Q.popleft()
17
18                  if p.node_id not in visited:
19                      # We check if the node is in the visited nodes list. If is
20                      # not in the list, we add it
21                      visited.append(p.node_id)
22
23                      ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
24                          p.node_id, p.parent_id, p.value)
25                      for child in p.children.values():
26                          Q.append(child)
27
28              return ret
29          return  traverse_tree(self)
30
31
32  if __name__ == '__main__':
33      # We add items to the tree
34      T = TreeBFS(0, 10)
35      T.add_node(1, 8, 0)
36      T.add_node(2, 12, 0)
37      T.add_node(3, 4, 1)
```

```
    T.add_node(4, 4, 1)
    T.add_node(5, 1, 3)
    T.add_node(6, 18, 2)

    print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 1, parent_id: 0 -> value: 8
node_id: 2, parent_id: 0 -> value: 12
node_id: 3, parent_id: 1 -> value: 4
node_id: 4, parent_id: 1 -> value: 4
node_id: 6, parent_id: 2 -> value: 18
node_id: 5, parent_id: 3 -> value: 1
```

Depth-First Search

In the **DFS** approach, the traversal algorithm starts from the top node and then goes down until it reaches a leaf. To achieve this kind of traversal, we have to use a stack to add the children nodes that we will visit in future iterations:

```
class TreeDFS(Tree):
    # We inherited the original class Tree from the previous example and
    # override the __repr__ method to apply the BFS algorithm.

    def __repr__(self):
        def traverse_tree(root):
            ret = ''
            Q = []
            Q.append(root)

            # We use a list to keep the visited nodes
            visited = []

            while len(Q) > 0:
                p = Q.pop()

                if p.node_id not in visited:
                    # We check if the node is in the visited nodes list. If is
```

```
19                      # not in the list, we add it
20                      visited.append(p.node_id)
21
22                      ret += "node_id: {}, parent_id: {} -> value: {}\n".format(
23                          p.node_id, p.parent_id, p.value)
24                      for child in p.children.values():
25                          Q.append(child)
26
27              return ret
28          return traverse_tree(self)
29
30
31  if __name__ == '__main__':
32      # We add items to the tree
33      T = TreeDFS(0, 10)
34      T.add_node(1, 8, 0)
35      T.add_node(2, 12, 0)
36      T.add_node(3, 4, 1)
37      T.add_node(4, 4, 1)
38      T.add_node(5, 1, 3)
39      T.add_node(6, 18, 2)
40
41      print(T)
```

```
node_id: 0, parent_id: None -> value: 10
node_id: 2, parent_id: 0 -> value: 12
node_id: 6, parent_id: 2 -> value: 18
node_id: 1, parent_id: 0 -> value: 8
node_id: 4, parent_id: 1 -> value: 4
node_id: 3, parent_id: 1 -> value: 4
node_id: 5, parent_id: 3 -> value: 1
```

In this chapter, we reviewed the most common data structures in Python, if the reader wants to go deeper into data structures and algorithms we recommend the books of Cormen [4] and Karumanchi [5].

2.3 Hands-On Activities

Activity 2.1

In a production line of bottles, you have to implement software that lets the user predict the *output* of his factory. A colleague has modeled the process, and he asks you to finish the code by adding all the functionalities. Your **task** is to complete only the methods that appear commented in the code, explained below.

```
1  from collections import deque
2  from package import Package
3  from bottle import Bottle
4
5
6  class Machine:
7
8      def process(self, incoming_production_line):
9          print("----------------------")
10         print("Machine {} started working.".format(
11             self.__class__.__name__))
12
13
14 class BottleModulator(Machine):
15
16     def __init__(self):
17         self.bottles_to_produce = 0
18
19     def process(self, incoming_production_line=None):
20         super().process(incoming_production_line)
21         # --------------
22         # Complete the method
23         # --------------
24         return None
25
26
27 class LowFAT32(Machine):
28
29     def __init__(self):
```

```
30          self.discarded_bottles = []
31
32      def discard_bottle(self, bottle):
33          self.discarded_bottles.append(bottle)
34
35      def print_discarded_bottles(self):
36          print("{} bottles were discarded".format(
37              len(self.discarded_bottles)))
38
39      def process(self, incoming_production_line):
40          # -----------------
41          # Complete the method
42          # -----------------
43          return None
44
45
46  class HashSoda9001(Machine):
47
48      def process(self, incoming_production_line):
49          super().process(incoming_production_line)
50          # ----------------
51          # Complete the method
52          # ----------------
53          return None
54
55
56  class PackageManager(Machine):
57
58      def process(self, incoming_production_line):
59          packages = deque()
60          for stack in incoming_production_line.values():
61              package = Package()
62              package.add_bottles(stack)
63              packages.append(package)
64          return packages
```

```
65
66
67  class Factory:
68
69      def __init__(self):
70          self.bottlemodulator = BottleModulator()
71          self.lowFAT32 = LowFAT32()
72          self.hashSoda9001 = HashSoda9001()
73          self.packageManager = PackageManager()
74
75      def producir(self, num_bottles):
76          self.bottlemodulator.bottles_to_produce = num_bottles
77          product = None
78          for machine in [self.bottlemodulator,
79                          self.lowFAT32,
80                          self.hashSoda9001,
81                          self.packageManager]:
82              product = machine.process(product)
83          return product
84
85
86  if __name__ == "__main__":
87
88      num_bottles = 423
89
90      factory = Factory()
91      output = factory.producir(num_bottles)
92      print("----------------------")
93      print("{} bottles produced {} packages".format(
94          num_bottles, len(output)))
95      for package in output:
96          package.see_content()
97      print("----------------------")
```

The class ***bottle*** has a parameter that denote its maximum capacity in liters (lts). By default the maximum capacity is 1 lt. and it is filled with the delicious soda *DCC-Cola*. The class ***factory*** has a method that receives the number of bottles

that will be produced. Each machine has the method ***process*** that receives bottles as ***incoming_production_line***.

1. **Bottlemodulator**: this machine creates the bottles. It has an attribute to set up the number of bottles to produce. The ***incoming_production_line*** is null. By default, it produces bottles of 1-liter capacity, however, after a particular number of bottles occurs the following variations:

   - Each five produced bottles, the next bottle (ex: 6, 11, 16, ...) will have three times of the standard capacity (1 lt.)
   - Each six produced bottles, the next bottle (ex: 7, 13, 19, ...) will have half of the last bottle capacity (1 lt.) plus four times of the antepenultimate bottle.

   At the end of this process, each bottle passes to a production line in the same order that they were created. The method that models this machine returns the production line.

2. **Low-FAT32**: this machine processes the production line provided by the **Bottlemodulator** and fills a dispatch line. It chooses the first bottle from the incoming production line. If there is no bottle in this new line, the machine adds the first one. If the machine already has added bottles:

   - It puts the bottle at the end of the dispatch line if the bottle has the same or greater capacity than the last bottle in the line.
   - It puts the bottle at the beginning of the dispatch line if the bottle has the same or less capacity that the first bottle in the line.
   - It discards the bottle in other cases.

   At the end of this process, it shows the number of discarded bottles. Finally, the model of the machine returns the dispatch line.

3. **HashSoda9001**: this machine classifies and stacks the incoming bottles according to their size (for **n** different capacities we will have **b** different stacks).

4. **PackageManager**: the function of this machine is to pack the stacks of bottles provided by the previous machine. It returns a list of packages. Your colleague has already programmed this machine.

You have to complete the BottleModulator, Low-FAT32, and HashSoda9001 machines, such that they return the expected output.

Activity 2.2

Most of the data structures saw in this chapter are linear, however, in some situations, we require to work using more dimensions, such as a graph to navigate in a labyrinth or a tree to make decisions or perform depth search. In this activity, we use a data structure that models a subway map, where each station can have until four adjacent subway stations: *Right*, *Left*, *Up*, and *Down*. Below we show an example of a map. Note that station one is connected to station two, but not the other way around. We can also note that it is possible to reach station eleven from station zero. However, it is not feasible to go from station zero to station nineteen.

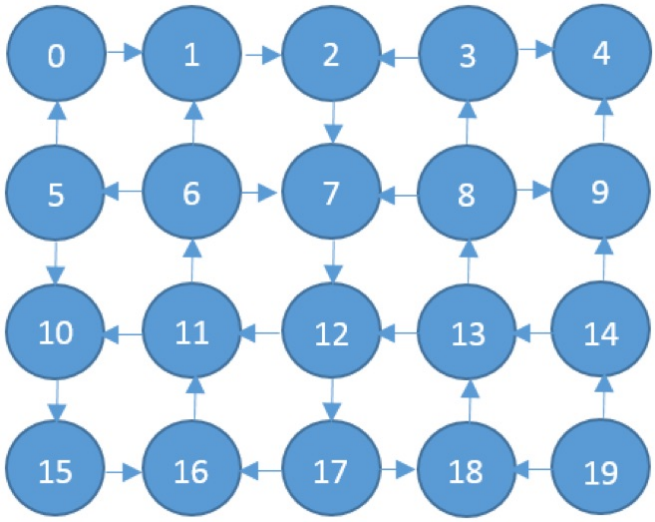


Figure 2.15

Get the files *main.py* and *station.py* provided in *https://github.com/advancedpythonprogramming*

Modify only the file *main.py* and complete the `path` method, that works as follows:

- This method receives two subway stations
- It returns **False** if it is not possible to arrive from the origin to the end. You must respect the rest of the paths.
- It returns **True** if exist a path between the origin and the destination. It also prints the path.

You can create methods, classes or anything you require in the *main.py* file. You cannot change the signature of the `path` method. You can work with the map provided in 2.15. The method used to generate the map is `SubwayMetro.example_map()`

Chapter 3

Functional Programming

In general, the most used programming approach for introductory courses is the procedural one, where we organize the code as a list of instructions that tell the computer how to process a given input. In chapter 1 we introduced the Object-Oriented paradigm, where programs represent the functionalities by objects-interaction through their attributes and methods that modify the attributes or states of each object. In the functional approach, we organize our code as a set of related functions that we can pass as arguments, modify or return them. Functions' outputs can be inputs to other functions. Functions' scope is only the code contained inside them; they do not use or modify any data outside their scope

The functional programming forces us to write a modular solution, breaking into apart our tasks into small pieces. It is advantageous during debugging and testing because the wrote functions are small and easy to read. In debugging, we can quickly isolate errors in a particular function. When we test a program, we see each function as a unit, so we only need to create the right input and then check its corresponding output.

Python is a multi-paradigm programming language, *i. e*, our solutions could be written simultaneously either in a procedural way, using object-oriented programming or applying a functional approach. In this chapter, we explain the core concepts of functional programming in Python and how we develop our applications using this technique.

3.1 Python Functions

There are many functions already implemented in Python, mainly to simplify and to abstract from calculations that we can apply to several different types of classes (duck typing). We recommend the reader to check the complete list of built-in functions in [1]. Let's see a few examples:

Len

Returns the number of elements in any container (list, array, set, etc.)

```
print(len([3, 4, 1, 5, 5, 2]))
print(len({'name': 'John', 'lastname': 'Smith'}))
print(len((4, 6, 2, 5, 6)))
```

```
6
2
5
```

This function comes implemented as the internal method (`__len__`) in most of Python default classes:

```
print([3, 4, 1, 5, 5, 2].__len__())
print({'name': 'John', 'lastname': 'Smith'}.__len__())
```

```
6
2
```

When `len(MyObject)` is called, it actually calls the method `MyObject.__len__()`:

```
print(id([3, 4, 1, 5, 5, 2].__len__()))
print(id(len([3, 4, 1, 5, 5, 2])))
```

```
4490937616
4490937616
```

We can also override the `__len__` method. Suppose we want to implement a special type of list (`MyList`) such that `len(MyList())` returns the length of the list ignoring repeated occurences:

```
from collections import defaultdict


class MyList(list):
    def __len__(self):
        # Each time this method is called with a non-existing key, the
        # key-value pair is generated with a default value of 0
        d = defaultdict(int)
        # This value comes from calling "int" without arguments. (Try
```

```
        # typing int() on Python's console)

        # Here we call the original method from the super-class list
        for i in range(list.__len__(self)):
            d.update({self[i]: d[self[i]] + 1})

        # Here we call d's (a defaultdict) len method
        return len(d)


L = MyList([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
print(len(L))
```

```
7
```

```
from collections import defaultdict

# Another way of achieving the same behaviour
class MyList2(list):
    def __len__(self):
        d = defaultdict(int)

        for i in self:  # Here we iterate over the items contained in the object
            d.update({i: d[i] + 1})

        return len(d)


L = MyList2([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
print(len(L))
```

```
7
```

```
# Yet another way
class MyList3(list):
    def __len__(self):
        d = set(self)
```

```
        return len(d)

L = MyList3([1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 7, 2, 2, 3, 3, 1, 1])
print(len(L))
```

```
7
```

Getitem

Declaring this function within a class allows each instance to become iterable (you can iterate over the object, soon we delve further into iterable objects). Besides allowing iteration, the `__getitem__` method lets us use indexation over the objects:

```
class MyClass:
    def __init__(self, word=None):
        self.word = word

    def __getitem__(self, i):
        return self.word[i]


p = MyClass("Hello World")
print(p[0])

[print(c) for c in p]

(a, b, c, d) = p[0:4]
print(a, b, c, d)
print(list(p))
print(tuple(p))
```

```
H
H
e
l
l
o
```

```
W
o
r
l
d
H e l l
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd')
```

Reversed

The `reversed` function takes a sequence as input and returns a copy of the sequence in reversed order. We can also customize the function by overriding the `__reversed__` method in each class. If we do not customize this function, the built-in will be used, by iterating once from `__len__` to `0` using the `__getitem__` method.

```
1  a_list = [1, 2, 3, 4, 5, 6]
2
3
4  class MySequence:
5      # given that we are not overriding the __reversed__ method, the built-in
6      # will be used (iterating with __getitem__ and __len___)
7      def __len__(self):
8          return 9
9
10     def __getitem__(self, index):
11         return "Item_{0}".format(index)
12
13
14 class MyReversed(MySequence):
15     def __reversed__(self):
16         return "Reversing!!"
17
18
19 for seq in a_list, MySequence(), MyReversed():
20     print("\n{} : ".format(seq.__class__.__name__), end="")
```

```
21      for item in reversed(seq):
22          print(item, end=", ")
```

```
list : 6, 5, 4, 3, 2, 1,
MySequence : Item_8, Item_7, Item_6, Item_5, Item_4, Item_3, Item_2, Item_1,
Item_0,
MyReversed : R, e, v, e, r, s, i, n, g, !, !,
```

Enumerate

The `enumerate` method creates an iterable of tuples, where the first item in each tuple is the index and the second is the original object in the corresponding index.

```
1  a_list = ["a", "b", "c", "d"]
2
3  for i, j in enumerate(a_list):
4      print("{}: {}".format(i, j))
5
6  print([pair for pair in enumerate(a_list)])
7
8  # We create a dictionary using the index given by "enumerate" as key
9  print({i: j for i, j in enumerate(a_list)})
```

```
0: a
1: b
2: c
3: d
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

Zip

This function takes n sequences (two or more) and generates a sequence of n-tuples with the relative objects in each sequence:

```
1  variables = ['name', 'lastname', 'email']
2  p1 = ["John", 'Smith', 'js1@hotmail.com']
```

```
p2 = ["Thomas", 'White', 'thwh@gmail.com']
p3 = ["Jeff", 'West', 'jwest@yahoo.com']

contacts = []
for p in p1,p2,p3:
    contact = zip(variables, p)
    contacts.append(dict(contact))

for c in contacts:
    print("Name: {name} {lastname}, email: {email}".format(**c))
    #**c passes the dictionary as a keyworded list of arguments
```

```
Name: John Smith, email: js1@hotmail.com
Name: Thomas White, email: thwh@gmail.com
Name: Jeff West, email: jwest@yahoo.com
```

The `zip` function is also its own inverse:

```
A = [1, 2, 3, 4]
B = ['a', 'b', 'c', 'd']

zipped = zip(A, B)
zipped = list(zipped)
print(zipped)
unzipped = zip(*zipped)
unzipped = list(unzipped)
print(unzipped)
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
[(1, 2, 3, 4), ('a', 'b', 'c', 'd')]
```

Comprehensions

Defining a set of elements by comprehension allows you to explicitly describe the content without enumerating each one of the elements. In Python we can do this as follows:

List comprehensions:

```
a_list = ['1', '4', '55', '65', '4', '15', '90']
int_list = [int(c) for c in a_list]
print("int_list:", int_list)

int_list_2d = [int(c) for c in a_list if len(c) > 1]
print("int_list_2d:", int_list_2d)
```

```
int_list: [1, 4, 55, 65, 4, 15, 90]
int_list_2d: [55, 65, 15, 90]
```

Sets and Dictionary comprehensions:

```
from collections import namedtuple

#namedtuple is a tuple subclass that has fields (with arbitrary names),
#which can be accessed as tuple.field
Movie = namedtuple("Movie", ["title", "director", "genre"])
movies = [Movie("Into the Woods", "Rob Marshall", "Adventure"),
          Movie("American Sniper", "Clint Eastwood", "Action"),
          Movie("Birdman", "Alejandro Gonzalez Inarritu", "Comedy"),
          Movie("Boyhood", "Richard Linklater", "Drama"),
          Movie("Taken 3", "Olivier Megaton", "Action"),
          Movie("The Imitation Game", "Morten Tyldum", "Biography"),
          Movie("Gone Girl", "David Fincher", "Drama")]

# set comprehension
action_directors = {b.director for b in movies if b.genre == 'Action'}
print(action_directors)
```

```
{'Clint Eastwood', 'Olivier Megaton'}
```

We can create dictionaries from search results:

```
action_directors_dict = {b.director: b for b in movies if b.genre == 'Action'}
print(action_directors_dict)
print(action_directors_dict['Olivier Megaton'])
```

```
{'Clint Eastwood': Movie(title='American Sniper', director='Clint Eastwood',
```

```
genre='Action'), 'Olivier Megaton': Movie(title='Taken 3',
director='Olivier Megaton', genre='Action')}
Movie(title='Taken 3', director='Olivier Megaton', genre='Action')
```

Iterables and Iterators

A *iterable* is any object over which you can iterate. Therefore, we can use any iterable on the right side of a *for* loop. We can iterate an infinite amount of times over an iterable, just like with lists. This type of objects must contain the `__iter__` method.

A *iterator* is an object that iterates over an iterable. These objects contain the `__next__` method, which will return the next element each time we call it. The object returned by the `__iter__` method must be an iterator. Let's see the following example:

```
x = [11, 32, 43]
for c in x:
    print(c)
print(x.__iter__)
next(x)  # Lists are not iterators
```

```
11
32
43
<method-wrapper '__iter__' of list object at 0x10bef2e48>
'list' object is not an iterator
```

As we can see above, list objects are not iterators, but we can get an iterator over a list by calling the `iter` method.

```
y = iter(x)  # equivalent to x.__iter__
print(next(y))
print(next(y))
print(next(y))
```

```
11
32
43
```

```
class Card:
```

```
    FACE_CARDS = {11: 'J', 12: 'Q', 13: 'K'}

    def __init__(self, value, suit):
        self.suit = suit
        self.value = value if value <= 10 else Card.FACE_CARDS[value]

    def __str__(self):
        return "%s %s" % (self.value, self.suit)


class Deck:
    def __init__(self):
        self.cards = []
        for s in ['Spades', 'Diamonds', 'Hearts', 'Clubs']:
            for v in range(1, 14):
                self.cards.append(Card(v, s))


for c in Deck().cards:
    print(c)
```

```
1 Spades
  ...
K Spades
1 Diamonds
  ...
K Diamonds
1 Hearts
  ...
K Hearts
1 Clubs
  ...
K Clubs
```

Even though a `Deck` instance contains many cards, we can not iterate directly over it, only over `Deck().cards` (which corresponds to a list, an iterable object). Suppose we want to iterate over `Deck()` directly. In order to do so,

we should define the `__iter__` method.

```
class Deck:
    def __init__(self):
        self.cards = []
        for p in ['Spades', 'Diamonds', 'Hearts', 'Clubs']:
            for n in range(1, 14):
                self.cartas.append(Card(n, p))

    def __iter__(self):
        return iter(self.cards)


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

```
1 Spades
  ...
K Spades
1 Diamonds
  ...
K Diamonds
1 Hearts
  ...
K Hearts
1 Clubs
  ...
K Clubs
```

Let's see an example of how to create an iterator:

```
class Fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1

    def __iter__(self):
        return self
```

```
8
9       def __next__(self):
10          value = self.curr
11          self.curr += self.prev
12          self.prev = value
13          return value
14
15
16  f = Fib()
17  N = 10
18  l = [next(f) for i in range(N)]
19  print(l)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Python's module "`itertools`" provides many iterators. Here are some examples:

```
1   import itertools
2
3   letters = ['a', 'b', 'c', 'd', 'e', 'f']
4   bools = [1, 0, 1, 0, 0, 1]
5   nums = [23, 20, 44, 32, 7, 12]
6   decimals = [0.1, 0.7, 0.4, 0.4, 0.5]
7
8   # Iterates indefinitely over letters.
9   colors = itertools.cycle(letters)
10  print(next(colors))
11  print(next(colors))
12  print(next(colors))
13  print(next(colors))
14  print(next(colors))
15  print(next(colors))
16  print(next(colors))
17  print(next(colors))
18  print(next(colors))
19  print(next(colors))
```

```
a
b
c
d
e
f
a
b
c
d
```

```
# Iterates across all the iterables in the arguments consecutively.
for i in itertools.chain(letters, bools, decimals):
    print(i, end=" ")
```

```
a b c d e f 1 0 1 0 0 1 0.1 0.7 0.4 0.4 0.5
```

```
# Iterates over the elements in letters according to the condition in bools.
for i in itertools.compress(letters, bools):
    print(i, end=" ")
```

```
a c f
```

Generators

Generators are a particular type of iterators; they allow us to iterate over sequences without the need to save them in a data structure, avoiding unnecessary memory usage. Once we finish the iteration over a generator, the generator disappears. It is useful when you want to perform calculations on sequences of numbers that only serve a purpose in a particular calculation. The syntax for creating generators is very similar to a list comprehension, but instead of using square brackets `[]`, we use parentheses `()`:

```
from sys import getsizeof

# using parenthesis indicates that we are creating a generator
a = (b for b in range(10))

print(getsizeof(a))
```

```
7
8  c = [b for b in range(10)]
9
10 # c uses more memory than a
11 print(getsizeof(c))
12
13 for b in a:
14     print(b)
15
16 print(sum(a)) # the sequence has disappeared
```

```
72
192
0
1
2
3
4
5
6
7
8
9
0
```

Example: Suppose that the archive `logs.txt` contains the following lines::

```
Abr 13, 2014 09:22:34
Jun 14, 2014 08:32:11
May 20, 2014 10:12:54
Dic 21, 2014 11:11:62
WARNING We are about to have a problem.
WARNING Second Warning!
WARNING This is a bug
WARNING Be careful
```

```
inname, outname = "logs.txt", "logs_out.txt"

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l.replace('WARNING', '') for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

The contents of `logs_out.txt` should read as goes:

```
We are about to have a problem.
Second Warning!
This is a bug
Be careful
```

Generator Functions

Python functions are also able to work as generators, through the use of the `yield` statement. `yield` replaces `return`, which besides being responsible for returning a value, it assures that the next function call will be executed starting from that point. In other words, we work with a method that once it "returns" a value through `yield`, it transfers the control back to the outer scope only temporarily, waiting for a successive call to "generate" more values. Calling a generator function creates a generator object. However, this does not start the execution of the function:

```
def dec_count(n):
    print("Counting down from {}".format(n))
    while n > 0:
        yield n
        n -= 1
```

The function is only executed once we call the generated object's `__next__` method:

```
x = dec_count(10)  # Note that this does not print anything
print("{}\n".format(x))  # here we are printing the object itself
y = dec_count(5)
print(next(y))
print(next(y))
print(next(y))
```

```
print(next(y))
```

```
<generator object dec_count at 0x1080464c8>

Counting down from 5
5
4
3
2
```

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b


f = fibonacci()
print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
g1 = [next(f) for i in range(10)]
print(g1)
g2 = (next(f) for i in range(10))
for a in g2:
    print(a)
```

```
1
1
2
3
5
8
[13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
```

```
import numpy as np

def maximum(values):
    temp_max = -np.infty
    for v in values:
        if v > temp_max:
            temp_max = v
        yield temp_max

elements = [10, 14, 7, 9, 12, 19, 33]
res = maximum(elements)
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))
print(next(res))  # we've run out of list elements!
```

```
10
14
14
14
14
```

```
19
33
```

We can also interact with a function by sending messages. The `send` method allows us to send a value to the generator. We can assign that value to a variable by using the `yield` statement. When we write `a = yield`, we are assigning to the variable `a` the value sent by the `send` method. When we write `a = yield b`[1], besides assigning the sent value to `a`, the function is returning the object `b`:

```
def mov_avg():
    print("Entering ...")
    total = float((yield))
    cont = 1
    print("total = {}".format(total))
    while True:
        print("While loop ...")
        # Here i receive the message and also the yield returns total/count
        i = yield total / cont
        cont += 1
        total += i
        print("i = {}".format(i))
        print("total = {}".format(total))
        print("cont = {}".format(cont))
```

Note that the code must run until the first `yield` in order to start accepting values through `send()`. Hence it is always necessary to call `next()` (or `send(None)`) after having created the generator to be able to start sending data:

```
m = mov_avg()
print("Entering to the first next")
next(m)  # We move to the first yield
print("Leaving the first next")
m.send(10)
print("Entering to send(5)")
m.send(5)
print("Entering to send(0)")
m.send(0)
```

[1] Use of parentheses around `yield b` may be needed if you want to operate over the sent value

```
10 print("Entering to second send(0)")
11 m.send(0)
12 print("Entering to send(20)")
13 m.send(20)
```

```
Entering to the first next
Entering ...
Leaving the first next
total = 10.0
While loop ...
Entering to send(5)
i = 5
total = 15.0
cont = 2
While loop ...
Entering to send(0)
i = 0
total = 15.0
cont = 3
While loop ...
Entering to second send(0)
i = 0
total = 15.0
cont = 4
While loop ...
Entering to send(20)
i = 20
total = 35.0
cont = 5
While loop ...
```

The following example shows how to perform the UNIX grep command by using a generator function.

```
1 def grep(pattern):
2     print("Searching for %s" % pattern)
3     while True:
4         line = yield
```

```
        if pattern in line:
            print(line)


o = grep("Hello")  # creating the object won't execute the function yet
next(o)  # Move on to the first yield, "Searching for ..." will be printed

o.send("This line contains Hello")
o.send("This line won't be printed")
o.send("This line will (because it contains Hello :) )")
o.send("This line won't be shown either")
```

```
Searching for Hello
This line contains Hello
This line will (because it contains Hello :) )
```

Lambda Functions

Lambda functions are short methods created "on the fly". Their expressions are always returned (no need for the "`return`" statement). Examples:

```
strings = ["ZZ", "YY", "bb", "aa"]
print("Simple sort:", sorted(strings))

# If we want to sort according to the lowercase values:
def lower(s):
    return s.lower()

print("Lower sort: ", sorted(strings, key=lower))

# The same result can be achieved with a lambda function:
print("Lambda sort:", sorted(strings, key=lambda s: s.lower()))
```

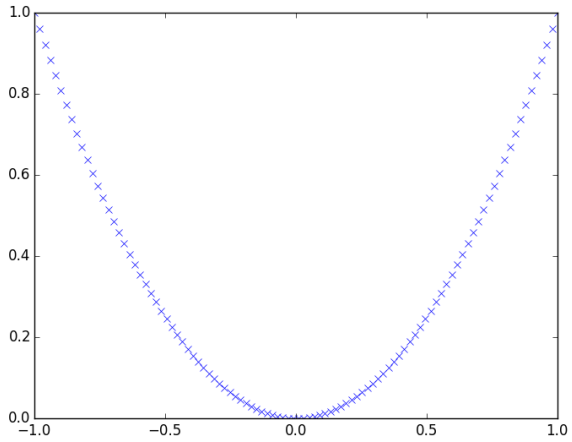
```
Simple sort: ['YY', 'ZZ', 'aa', 'bb']
Lower sort:  ['aa', 'bb', 'YY', 'ZZ']
Lambda sort: ['aa', 'bb', 'YY', 'ZZ']
```

Map

The `map` function takes a function and an iterable and returns a generator that results from applying the function to each value on the iterable. `map(f, iterable)` is equivalent to `[f(x) for x in iterable]`

```
from matplotlib import pyplot as plt
import numpy as np

pow2 = lambda x: x ** 2
# Creates a 100 element numpy array, ranging evenly from -1 to 1
t = np.linspace(-1., 1., 100)
plt.plot(t, list(map(pow2, t)), 'xb')
plt.show()
```



We can also apply `map` to more than one iterable at once:

```
a = [1, 2, 3, 4]
b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]

c1 = list(map(lambda x, y: x + y, a, b))

c2 = list(map(lambda x, y, z: x + y + z, a, b, c))

c3 = list(map(lambda x, y, z: 2.5 * x + 2 * y - z, a, b, c))

```

```
print(c1)
print(c2)
print(c3)
```

```
[18, 14, 14, 14]
[17, 10, 19, 23]
[37.5, 33.0, 24.5, 21.0]
```

Filter

`filter(f, sequence)` returns a new sequence that includes all the values from the original sequence in which the result of applying `f(value)` was `True`. Function `f` should always return a boolean value:

```
f = fibonacci() # Defined before
fib = [next(f) for i in range(11)]
odds = list(filter(lambda x: x % 2 != 0, fib))
print("Odd:", odds)

even = list(filter(lambda x: x % 2 == 0, fib))
print("Even:", even)
```

```
Odd: [1, 1, 3, 5, 13, 21, 55, 89]
Even: [2, 8, 34]
```

Reduce

`reduce(f, [s1,s2,s3,...,sn])` returns the result of applying `f` over the sequence `[s1,s2,s3,...,sn]` as follows: `f(f(f(f(s1,s2),s3),s4),s5),...` The following code shows an example:

```
from functools import reduce
import datetime

reduce(lambda x, y: x + y, range(1, 10))

# Lets compute the length of a file's longest line
# rstrip returns a string copy that has no trailing spaces
#(or the character specified)
```

```
9  # eg: "Hello...".rstrip(".") returns "Hello"
10
11 t = datetime.datetime.now()
12 r1 = reduce(max, map(lambda l: len(l.rstrip()), [line for line
13                                                  in open('logs_out.txt')]))
14 print("reduce time: {}".format(datetime.datetime.now() - t))
15 print(r1)
16
17 # Another way of doing the same with generator comprehensions and numpy
18 t = datetime.datetime.now()
19 r2 = max((len(line.rstrip()) for line in open('logs_out.txt')))
20 print("max(generator) time: {}".format(datetime.datetime.now() - t))
21 print(r2)
22
23 # To visualize the lines of the file
24 for line in open('logs_out.txt'):
25     print(line.rstrip())
```

```
reduce time: 0:00:00.000224
31
max(generator) time: 0:00:00.000158
31
We are about to have a problem.
Second Warning!
This is a bug
Be careful
```

3.2 Decorators

Decorators allow us to take an already implemented feature, add some behavior or additional data and return a new function. We can see decorators as functions that receive any function `f1` and return a new function `f2` with a modified behaviour. If our decorator is called `dec_1`, in order to modify a function and assign it to the same name, we should simply write `f1 = dec_1(f1)`.

Our function `f1` now contains the new data and aggregate behavior. One benefit of decorators is that we avoid the need to modify the code of the original function (and if we want the original version of the function, we simply remove the call to the decorator). It also avoids creating a different function with a different name (this would imply modifying all the calls to the function you want to change).

Take the following inefficient recursive implementation of a function that returns the Fibonacci numbers:

```
import datetime


def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


n = 35
t1 = datetime.datetime.now()
print(fib(n))
print("Execution time: {}".format(datetime.datetime.now() - t1))
```

```
9227465
Execution time: 0:00:06.462758
```

A more efficient implementation might try to "store" numbers already calculated in the Fibonacci sequence. We can use a decorator that receives the `fib` function, adds memory to it and checks for the existence of that number in a previous call:

```
def efficient_fib(f):  # recieves a function as an argument
    data = {}

    def func(x):  # this is the new function to be returned
        if x not in data:
            data[x] = f(x)  # the function recieved as an argument
```

```
7                                #is now called
8           return data[x]
9
10      return func
11
12  # we use the decorator.
13  fib = efficient_fib(fib)
14  # The fib function is now "decorated" by the function
15  #"efficient_fib"
16  t1 = datetime.datetime.now()
17
18  # We still use the same function name, there is no need
19  #to call the new function
20  print(fib(n))
21  print("Execution time: {}".format(datetime.datetime.now() - t1))
```

```
9227465
Execution time: 0:00:00.000144
```

Using Python's alternative notation for decorators:

```
1   @eficient_fib
2   def fib(n):
3       if n == 0:
4           return 0
5       elif n == 1:
6           return 1
7       else:
8           return fib(n-1) + fib(n-2)
9
10  n = 35
11  t1 = datetime.datetime.now()
12  print(fib(n))
13  print("Execution time: {}".format(datetime.datetime.now()-t1))
```

```
9227465
Execution time: 0:00:00.000038
```

We can use a hierarchy of decorators and receive parameters for decoration. A generic way to do this is:

```
def mydecorator(function):
    def _mydecorator(*args, **kw):
        # Do stuff here before calling the original function
        # call the function
        res = function(*args, **kw)
        # Do more stuff after calling the function
        return res

    # return the sub-function
    return _mydecorator
```

```
import time
import hashlib
import pickle

cache = {}


def is_obsolete(entry, duration):
    return time.time() - entry['time'] > duration


def compute_key(function, args, kw):

    key = pickle.dumps((function.__name__, args, kw))
    # returns the pickle representation of an object as a byte object
    # instead of writing it on a file

    # creates a key from the "frozen" key generated in the last step
    return hashlib.sha1(key).hexdigest()


def memoize(duration=10):
    def _memoize(function):
        def __memoize(*args, **kw):
```

```
            key = compute_key(function, args, kw)

            # do we have the value on cache?
            if key in cache and not is_obsolete(cache[key], duration):
                print('we already have the value')
                return cache[key]['value']

            # if we didn't
            print('calculating...')
            result = function(*args, **kw)
            # storing the result
            cache[key] = {'value': result, 'time': time.time()}
            return result

        return __memoize

    return _memoize
```

```
@memoize(0.0001)
def complex_process(a, b):
    return a + b

# This is the same as calling
# complex_process = memoize(0.0001)(complex_process)
# after defining the function

print(complex_process(2, 2))
print(complex_process(2, 1))
print(complex_process(2, 2))
print(complex_process(2, 2))
print(complex_process(2, 2))
print(complex_process(2, 2))
print(complex_process(2, 2))
print(complex_process(2, 2))
print(complex_process(2, 2))
```

```
calculating...
4
calculating...
3
we already have the value
4
we already have the value
4
we already have the value
4
calculating...
4
we already have the value
4
we already have the value
4
we already have the value
4
```

Here an example of an access protection decorator:

```
class User:
    def __init__(self, roles):
        self.roles = roles


class Unauthorized(Exception):
    pass


def protect(role):
    def _protect(function):
        def __protect(*args, **kw):
            user = globals().get('user')
            if user is None or role not in user.roles:
                raise Unauthorized("Not telling you!!")  # exceptions coming soon!
```

```
16              return function(*args, **kw)
17
18          return __protect
19
20      return _protect
21
22
23  john = User(('admin', 'user'))
24  peter = User(('user',))
25
26
27  class Secret:
28      @protect('admin')
29      def pisco_sour_recipe(self):
30          print('Use lots of pisco!')
31
32
33  s = Secret()
34  user = john
35  s.pisco_sour_recipe()
36  user = peter
37  s.pisco_sour_recipe()
```

```
Use lots of pisco!
__main__.Unauthorized: Not telling you!!
```

We can also decorate classes in the same way we decorate functions:

```
1  # Lets suppose we want to decorate a class such that it prints a warning
2  # when we try to spend more than what we've got
3  def add_warning(cls):
4      prev_spend = getattr(cls, 'spend')
5
6      def new_spend(self, money):
7          if money > self.money:
8              print("You are spending more than what you have, "
9                    "a debt has been generated in your account!!")
```

```
10          prev_spend(self, money)
11
12      setattr(cls, 'spend', new_spend)
13      return cls
14
15
16  @add_warning
17  class Buy:
18      def __init__(self, money):
19          self.money = money
20          self.debt = 0
21
22      def spend(self, money):
23          self.money -= money
24          if self.money < 0:
25              self.debt = abs(self.money)  # the debt is considered positive
26              self.money = 0
27          print("Current Balance = {}".format(self.money))
28
29
30  b = Buy(1000)
31  b.spend(1200)
```

```
You are spending more than what you have, a debt has been generated in your account!!
Current Balance = 0
```

3.3 Hands-On Activities

Activity 3.1

In this activity, we have a file called *Report.txt* contains information about patients that attended to the city hospital during one year. Each line refers to the fields *year of attention*, *month of attention*, *day of the week*, *assigned color*, *time of attention* and *release reason*, *separated by tabs*. The *assigned color* shows how critical is the medical condition of the patient. The color code is from most to less critical: *blue*, *red*, *orange*, *yellow* and *green*. Your task is to create an application able to read the information from the file and generate the necessary classes and objects using this information. To read the file, you have to create a *generator function* that *yields* each line of the file one by one. You

will also have to create a class called `Report`. An instance of this class has to keep the list of all the patients. The instance of `Report` has to be a *iterable*, such that, iterations over it must return each patient in its list of patients. Also, the `Report` class have to contain a function that given a color it returns all the patients assigned to this color. The returned list must be created using comprehension.

Each patient instance have all the information contained in the file *Report.txt* plus a personal `id` generated by using a *generator* function. Also, each patient must be able to be printed showing all his personal information, including the *id* assigned by the system. After reading all the file, your application must print out all the patients.

Activity 3.2

A soccer team needs to hire some new players to improve their results in the next sports season. They bought a data file called `players.txt` that contains information with most of the soccer players in the league. The team asks your help to process the data file and get valuable information for hiring. Each line of the archive contains information about one player in comma separated values format (CSV):

```
names; last_name_1; last_name_2; country; footedness; birth_day; birth_month;
                birth_year; number_of_goals; high_cm, weight_kg
```

Using mostly only **map**, **reduce** and **filter**, you must perform the following tasks:

1. Read the data file `players.txt` using `map`. Generate a list of tuples, where each tuple contains the data from each player.
2. For each of the queries below, create a function `name_query(list_tuples)` that returns the following:
   a) **Has the name:** Returns a list of tuples with the information about the players that have a defined name or last name (1 or 2)
   b) **Lefty-Chileans:** Returns a list of tuples with the information about all the lefty players from Chile.
   c) **Ages:** Returns a list of tuples with the format `(names, last_name_1, age)` of every player. For simplicity, just use the year of birth to calculate the age.
   d) **Sub-17:** Returns a list of tuples with the format `(names, last_name_1)` if every players that have 17 years old or less (hint: use the previous result).
   e) **Top Scorer:** Returns a tuple with all the information from the top scorer player. You can assume that exists just one top scorer.

f) **Highest obesity risk:** Returns a tuple with the format `(names, last_name_1, high_cm, weight_kg, bmi)`. Where `bmi` is the body mass index, calculated as the body mass divided by the square of the body height (kg/m^2)

Using all the coded functions, print the results of all the queries.

Activity 3.3

The owner of a hamburger store wants you to implement an upgrade for the current production management system. They have the main class that models each product; you have to create a decorator for that class such that you save every newly created instance in a list (belonging to that class) called `instances`.

The upgrade has to allow the system to compare the produced hamburgers according to a customizable attribute that may vary in the future. Your goal is to include a decorator called `compare_by` that receives as a parameter the name of the attribute used for the comparison, such that it allows for comparing instances. We have to make all the possible comparisons though the operators: $<, >, =, \leq, \geq$. For example, `hamburger1` $>$ `hamburger2` returns `True` if the attribute specified in the decorator for hamburger1 is higher than the same attribute in hamburger2.

The current function used by the management system to calculate the final price uses a fix tax value. However, The Government will change the tax percentage in the future from 19% to 23%. Your upgrade should also change the way the system calculates the amount of the tax applied to each sale. Unfortunately, the function cannot be directly modified. Therefore, one of the most suitable solutions is to change the behavior of the function using a decorator, called `change_tax`.

Tips and examples

- **Retrieving and modifying attributes**

  You can use `getattr` and `setattr` to retrieve and to update the attributes of an object.

```
# Access
old_value = getattr(object, 'attribute_name')
# Modify
setattr(object, 'attribute_name', new_value)
```

- **Decorating a class**

  The following example shows a decorator that modifies a class method such that it prints out a message every time we call it.

```
def call_alert(method_name):
    def _decorator(cls):
        method = getattr(cls, method_name)

        def new_method(*args, **kwargs):
            print('Calling the method!')
            return method(*args, **kwargs)

        setattr(cls, method_name, new_method)
        return cls

    return _decorator

#Here we apply it to a test class:
@call_alert('walk')
class Test:
    def walk(self):
        return 'I am walking'

if __name__ == "__main__":
    t = Test()
    print(t.walk())
```

The following script shows the current production management system used by the store. Add your decorators at the beginning and then, decorate the class `Hamburger` and the function `price_after_tax`:

```
class Hamburger:

    def __init__(self, high, diameter, meat_quantity):
        self.high = high
        self.diameter = diameter
        self.meat_quantity = meat_quantity

    def __repr__(self):
        return ('Hamburger {0} cms high, '
                '{1} cm of diameter and '
```

```
                '{2} meat quantity').format(self.high, self.diameter,
                                            self.meat_quantity)

def price_after_tax(price_before_tax):
    return (price_before_tax * 1.19 + 100)


if __name__ == "__main__":
    hamburger1 = Hamburger(10, 15, 2)
    hamburger2 = Hamburger(7, 10, 3)
    hamburger3 = Hamburger(10, 9, 2)

    print(hamburger2 > hamburger1)
    print(hamburger2 == hamburger3)
    print(hamburger1 < hamburger3)

    print(Hamburger.instances)
    hamburger4 = Hamburger(12, 20, 4)
    print(Hamburger.instances)
    print(price_after_tax(2000))
```

Chapter 4

Meta Classes

Python classes are also objects, with the particularity that these can create other objects (their instances). Since classes are objects, we can assign them to variables, copy them, add attributes, pass them as parameters to a function, etc.

```
class ObjectCreator:
    pass

print(ObjectCreator)


def visualize(o):
    print(o)

visualize(ObjectCreator)
```

```
<class 'ObjectCreator'>
<class 'ObjectCreator'>
```

```
# Here we check if ObjectCreator has the attribute weight
print(hasattr(ObjectCreator, 'weight'))
```

```
False
```

```
# Here we are directly adding the weight attribute
ObjectCreator.weight = 80
print(hasattr(ObjectCreator, 'weight'))
print(ObjectCreator.weight)
```

```
True
80
```

```
# Assigning the class to a new variable
# Note that both variables reference the same object
ObjectCreatorMirror = ObjectCreator
print(id(ObjectCreatorMirror))
print(id(ObjectCreator))
print(ObjectCreatorMirror.weight)
```

```
140595089871608
140595089871608
80
```

Note that any changes we make to a class affect all of the class objects, including those that were already instantiated:

```
class Example:
    pass

x = Example()
print(hasattr(x, 'attr'))
Example.attr = 33
y = Example()
print(y.attr)
Example.attr2 = 54
print(y.attr2)
Example.method = lambda self: "Calling Method..."
print(y.method())
print(hasattr(x, 'attr'))
```

```
False
33
54
Calling Method...
True
```

4.1 Creating classes dynamically

Since classes are objects, we can create them at runtime just like any other object. For example, you can create a class within a function using the `class` statement:

```
def create_class(name):
    if name == 'MyClass':
        class MyClass:  # Usual way of creating a class
            pass
        return MyClass
    else:
        class OtherClass:
            pass
        return OtherClass

c1 = create_class('MyClass')
print(c1())
```

```
<MyClass object at 0x1078ff710>
```

We could also create a class in runtime using Python's `exec` command, which runs the code written in the input string. (You should be extremely careful with this function, and never execute a user given code, as it may contain malicious instructions).

```
name = "MyClass"
my_class = """
class %s():
    def __init__(self, a):
        self.at = a
""" % (name)
exec(my_class)
e = MyClass(8)
print(e.at)
```

```
8
```

That's pretty much, more of the same we have done so far. Now let's do it dynamically. First, let's remember that the `type` function returns an object's type:

```
print(type(1))
print(type("1"))
print(type(c1))
print(type(c1()))
# type is also an object of type 'type', it is an instance of itself
print(type(type))
```

```
<class 'int'>
<class 'str'>
<class 'type'>
<class 'MyClass'>
<class 'type'>
```

`type` can also create objects in runtime by taking a class descriptors as parameters. In other words, if we call `type` with only one argument, we are asking the type of the argument, but if you call it with three arguments, we are asking for the creation of a class. The first argument is the class name; the second argument is a tuple that contains all the parent classes. Finally, the third argument is a dictionary that contains all the class's attributes and methods: `{attr_name:attr_value}` or `{method_name:function}`. Below we show an example:

```
name = "MyClass"
c2 = type(name, (), {})
```

```
# We can do the same with a function
def create_class(name):
    c = type(name, (), {})
    return c

# Here we create the class MyClass2
create_class("MyClass2")()
```

Obviously we can also add attributes:

```
def create_class(name, attr_name, attr_value):
    return type(name, (), {attr_name: attr_value})

Body = create_class("Body", "weight", 100)
bd = Body()  # using it as a normal class to create instances.
print(bd.weight)
```

```
100
```

We can also add functions to the class dictionary, to create the methods of the class:

```
# a function that will be used as a method in the class we shall create
def lose_weight(self, x):
    self.weight -= x

Body = type("Body", (), {"weight": 100, "lose_weight": lose_weight})
bd = Body()

print(bd.weight)
bd.lose_weight(10)
print(bd.weight)
```

```
100
90
```

To inherit from the `Body` class:

```
class MyBody(Body):
    pass
```

we should write:

```
MyBody = type("MyBody", (Body,), {})
print(MyBody)
print(MyBody.weight)
```

```
<class 'MyBody'>
100
```

If we want to add methods to `MyBody`:

```
def see_weight(self):
    print(self.weight)

MyBody = type("MyBody", (Body,), {"see_weight": see_weight})
print(hasattr(Body, "see_weight"))
print(hasattr(MyBody, "see_weight"))
```

```
print(getattr(MyBody, "see_weight"))
print(getattr(MyBody(), "see_weight"))

m1 = MyBody()
m1.see_weight()
```

```
False
True
<function see_weight at 0x1078e02f0>
<bound method MyBody.see_weight of <MyBody object at 0x1078ffc50>>
100
```

4.2 Metaclasses

Metaclasses are Python's class creators; they are the classes of classes. `type` is Python's metaclass by default. It is written in lower_case to maintain consistency with `str`, the class that creates string objects, and with `int`, the class that creates objects of integer type. `type` is simply the class that creates objects of type `class`.

In Python all objects are created from a class:

```
height = 180
print(height.__class__)
name = "Carl"
print(name.__class__)


def func(): pass
print(func.__class__)


class MyClass():
    pass
print(MyClass.__class__)
```

```
<class 'int'>
<class 'str'>
<class 'function'>
```

```
<class 'type'>
```

We can also check what is the creator class of all the previous classes:

```
print(height.__class__.__class__)
print(name.__class__.__class__)
print(func.__class__.__class__)
print(MyClass.__class__.__class__)
```

```
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
```

"metaclass" keyword argument in base classes

We can add the `metaclass` keyword in the list of keyword arguments of a class. If we do it, Python uses that metaclass to create the class; otherwise, Python will use `type` to create it:

```
class MyBody(Body):
    pass


class MyOtherBody(Body, metaclass=type):
    pass
```

Python asks if the `metaclass` keyword is defined within `MyBody` class arguments. If the answer is "yes", like in `MyOtherBody`, a class with that name is created in memory using the value of `metaclass` as a creator. If the answer is "no", Python will use the same metaclass of the parent class to create the new class. In the case of `MyBody`, the metaclass used is Body's metaclass i.e: `type`. What can we put in `metaclass`?: Anything that can create a class. In Python, `type` or any object that inherits from it can create a class.

Personalized Metaclasses

Before we start explaining of to personalize a metaclass, we will take a look at the structure of regular Python classes we have been using so far:

```
class System:
```

```
    # users_dict = {}  we will do this automatically inside __new__

    # cls is the object that represents the class
    def __new__(cls, *args, **kwargs):
        cls.users_dict = {}
        cls.id_ = cls.generate_user_id()
        # object has to create the class (everything inherits from object)
        return super().__new__(cls)

    # recall that self is the object that represents the instance of the class
    def __init__(self, name):
        self.name = name

    def __call__(self, *args, **kwargs):
        return [System.users_dict[ar] for ar in args]

    @staticmethod
    def generate_user_id():
        count = 0
        while True:
            yield count
            count += 1

    def add_user(self, name):
        System.users_dict[name] = next(System.id_)


if __name__ == "__main__":
    e = System("Zoni")
    e.add_user("KP")
    e.add_user("CP")
    e.add_user("BS")
    print(e.users_dict)
    print(e("KP", "CP", "BS"))
    print(System.mro())  # prints the class and superclasses
```

```
{'KP': 0, 'CP': 1, 'BS': 2}
[0, 1, 2]
[<class '__main__.System'>, <class 'object'>]
```

The __new__ method is in charge of the construction of the class. `cls` corresponds to the object that represents the created class. Any modification we want to do in the class before its creation can be done inside the __new__ method. In the example above, we are creating a dictionary (`users_dict`) and an id (`id_`). Both of them will belong to the class (static), not to the instances of the class. Note that __new__ **has to return the created class**, in this case returning the result of the __new__ method of the superclass.

Inside __init__, **the class is already created**. Now the main goal is to initialize the instances of it, by modifying `self`, the object that represents the instance of the class. In the example above, the instance initialization just registers the variable `name` inside the instance (`self.name = name`).

Finally, the __call__ method is in charge of the action that will be performed every time an instance of the class is called with parenthesis (treated as a callable). In the example, when we execute `e("KP", "CP", "BS")`, we are executing `e.__call__` with the passed arguments.

Now we are ready to understand how to personalize a metaclass. Following the same structure of regular Python classes mentioned above, imagine that the class now is a metaclass, and the instance is a class. In other words, instead of `cls` we use `mcs` in the __new__ method and instead of `self` we use `cls` in the __init__ method. The __call__ method will be in charge of the action performed when an instance of the metaclass (i.e. the class) is called with parenthesis.

The primary purpose of metaclasses is to change a class automatically during its creation. To control the creation and initialization of a class, we can implement the __new__ and __init__ methods in the metaclass (overriding). We must implement __new__: when we want to control the creation of a new object (class); and __init__: when we want to control the object initialization (in this context a class) after its creation.

```
1 class MyMetaClass(type):
2
3     def __new__(meta, clsname, bases, clsdict):
4         print('-----------------------------------')
5         print("Creating Class: {} ".format(clsname))
6         print(meta)
7         print(bases)
8         # Suppose we want to have a mandatory attribute
```

```
9           clsdict.update(dict({'mandatory_attribute': 10}))
10          print(clsdict)
11          return super().__new__(meta, clsname, bases, clsdict)
12          # we are calling 'type' __new__ method after doing the desired
13          # modifications. Note hat this method is the one that would have
14          # been called had we not used this personalized metaclass
15
16
17  class MyClass(metaclass=MyMetaClass):
18
19      def func(self, params):
20          pass
21
22      my_param = 4
23
24  m1 = MyClass()
25  print(m1.mandatory_attribute)
```

```
-----------------------------------
Creating Class: MyClass
<class 'MyMetaClass'>
()
{'mandatory_attribute': 10, 'my_param': 4, '__qualname__': 'MyClass',
'func': <function MyClass.func at 0x1078e0620>, '__module__': 'builtins'}
10
```

Overwriting the __call__ method

The `__call__` method is executed each time the **already created** class is **called** to instantiate a new object. Here is an example of how the `__call__` method can be intercepted whenever an object is instantiated:

```
1  class MyMetaClass(type):
2
3      def __call__(cls, *args, **kwargs):
4          print("__call__ of  {}".format(str(cls)))
5          print("__call__ *args= {}".format(str(args)))
6          return super().__call__(*args, **kwargs)
```

```
7
8
9  class MyClass(metaclass=MyMetaClass):
10
11     def __init__(self, a, b):
12         print("MyClass object with a=%s, b=%s" % (a, b))
13
14 print('creating a new object...')
15 obj1 = MyClass(1, 2)
```

```
creating a new object...
__call__ of  <class 'MyClass'>
__call__ *args= (1, 2)
MyClass object with a=1, b=2
```

Overwriting the __init__ method

We can also override the `__init__` method to mimic the behavior of the previous example. The main difference is that `__init__` (just like `__new__`) is called upon when creating the class, however `__call__` is called when creating a new instance:

```
1  class MyMetaClass(type):
2
3      def __init__(cls, name, bases, dic):
4          print("__init__ of  {}".format(str(cls)))
5          super().__init__(name, bases, dic)
6
7
8  class MyClass(metaclass=MyMetaClass):
9
10     def __init__(self, a, b):
11         print("MyClass object with a=%s, b=%s" % (a, b))
12
13 print('creating a new object...')
14 obj1 = MyClass(1, 2)
```

```
__init__ of  <class 'MyClass'>
```

```
creating a new object...
MyClass object with a=1, b=2
```

4.3 Hands-On Activities

Activity 4

In the file called *AC04_0_provided_code.py* we have two implemented classes and a *main*. You have to create a metaclass called `MetaRobot` that must add the following data and methods to the `Robot` class:

- The `creator` (static) variable: It must be your user id
- The `start_ip` (static) variable: It is the IP address from where the robot is initialized. The address is "190.102.62.283"
- The `check_creator` method: This method verifies that the robot exists inside the list of programmers. I must print out a message indicating if the creator is inside the programmer's list or not.
- The `disconnect` method: By using this method, the robot can disconnect any hacker that is on the same port as the robot. In case the robot finds a hacker in the same port, it must print out a message telling the situation. Assume that the port's `hacker` attribute has to be changed to 0 to disconnect it.
- The `change_node` method: With this method, the robot can modify the node (port) to anyone that gets inside the network. It must print out a message indicating from what node it is coming from and what is its destination.

Consider that only the `Robot` class can be builded from the `MetaRobot` metaclass. In case any other class is attempted to be created from `MetaRobot` you should raise an error. It is forbiden to modify the *AC04_0_provided_code.py* file, everything has to be done through the `MetaRobot` metaclass.

Chapter 5

Exceptions

Exceptions are execution errors or situations where the program cannot obtain a valid or expected value. In most cases, they are indicating that something is going wrong with the execution of the program. For example, this situation happens when the input data type does not match with the expected by our program, or any other runtime errors. These conditions produce the unexpected termination of the execution. Python represents exceptions as objects.

5.1 Exception Types

In Python, all exceptions inherit from the `BaseException` class. Below, we see examples of the most common Python exceptions.

```
1  # 0.py
2
3  # SyntaxError exception: print is valid Python2.x, but incorrect in Python3.x
4  print 'Hello World'
```

```
  File "0.py", line 4
    print 'Hello World'
                       ^
SyntaxError: Missing parentheses in call to 'print'
```

```
1  # 1.py
2
3  # In this Python version print is a function
4  # and requires params inside the brackets
```

```
5 print('Hello World')
```

```
Hello World
```

```
1 # 2.py
2
3 # NameError exception: for data input in Python2.x, but incorrect in
4 # Python3.x
5 a = raw_input('Enter a number: ')
6 print(a)
```

```
Traceback (most recent call last):
  File "2.py", line 4, in <module>
    a = raw_input('Enter a number: ')
NameError: name 'raw_input' is not defined
```

```
1 # 3.py
2
3 # ZeroDivisionError exception: division by zero
4 x = 5.0 / 0
```

```
Traceback (most recent call last):
  File "4.py", line 5, in <module>
    x = 5.0 / 0
ZeroDivisionError: float division by zero
```

```
1 # 4.py
2
3 # IndexError exception: index out of range. A typical error that ocurrs
4 # when we try to access to an element of a list with an index that exceeds
5 # its size.
6 # Lists in Python have indexes from 0 to len(list_)-1
7
8 age = [36, 23, 12]
9 print(age[3])
```

```
Traceback (most recent call last):
  File "5.py", line 9, in <module>
```

```
    print(age[3])
IndexError: list index out of range
```

```
# 5.py

# TypeError exception: erroneus data type handling.
# A typical example is trying to concatenate a list
# with a variable that is not a list

age = [36, 23, 12]
print(age + 2)
```

```
Traceback (most recent call last):
  File "6.py", line 8, in <module>
    print(age + 2)
TypeError: can only concatenate list (not "int") to list
```

```
# 6.py

# Correct data type handling.
# To concatenate a list to another object, the latter has to be a list too

age = [36, 23, 12]
print(age + [2])
```

```
[36, 23, 12, 2]
```

```
# 7.py

# AttributeError exception: incorrect use of methods of a class or data type.
# In this example the class Car has only defined the method move, but the
# program tries execute the method stop() that doesn't exist


class Car:
    def __init__(self, doors=4):
        self.doors = doors
```

```
11
12      def mover(self):
13          print('avanzando')
14
15  chevi = Car()
16  chevi.stop()
```

```
Traceback (most recent call last):
  File "8.py", line 16, in <module>
    chevi.stop()
AttributeError: 'Car' object has no attribute 'stop'
```

```
1  # 8.py
2
3  # KeyError exception: incorrect use of key in dictionaries.
4  # In this example the program ask for an item associated with a key that
5  # doesn't appears in the dictionary
6
7  book = {'author': 'Bob Doe', 'pages': 'a lot'}
8  print(book['editorial'])
```

```
Traceback (most recent call last):
  File "9.py", line 8, in <module>
    print(book['editorial'])
KeyError: 'editorial'
```

5.2 Raising exceptions

We trigger exceptions in a program, or within a class or function using the statement `raise`. We can also add optionally a descriptive message to be shown when the exception is raised:

```
1  # 9.py
2
3
4  class Operations:
5
6      @staticmethod
```

```
7       def divide(num, den):
8           if den == 0:
9               # Here we generate the exception and we include
10              # information about its meaning.
11              raise ZeroDivisionError('Denominator is 0')
12          return float(num) / float(den)
13
14
15  print(Operations().divide(3, 4))
16  print(Operations().divide(3, 0))
```

```
0.75
Traceback (most recent call last):
  File "10.py", line 16, in <module>
    print(Operations().divide(3, 0))
  File "10.py", line 11, in divide
    raise ZeroDivisionError('Denominator is 0')
ZeroDivisionError: Denominator is 0
```

When an exception occurs inside a function contained in another one, this exception interrupts all the functions that include it and the main program. Besides the existent exceptions, it is possible to raise exceptions with customized messages:

```
1   # 10.py
2
3
4   class Circle:
5
6       def __init__(self, center):
7           if not isinstance(center, tuple):
8               raise Exception('Center has to be a tuple')
9               print('This line is not printed')
10
11          self.center = center
12
13      def __repr__(self):
14          return 'The center is {0}'.format(self.center)
```

```
15
16
17 c1 = Circle((2, 3))
18 print(c1)
19
20 c2 = Circle([2, 3])
21 print(c2)
```

```
The center is (2, 3)
Traceback (most recent call last):
  File "11.py", line 20, in <module>
    c2 = Circle([2, 3])
  File "11.py", line 8, in __init__
    raise Exception('Center has to be a tuple')
Exception: Center has to be a tuple
```

5.3 Exception handling

Every time an exception occurs, it is possible to handle it with the statements `try` and `except`. When a block of instructions defined inside the *try* statement triggers an exception, and the `except` processed it using the instructions inside of it. Afterward, the program continues its execution normally and does not stop or crash. The block of instructions inside the `except` defines how the program behaves depending on the exception type. In the following example, we observe that the program does not crash, even though occurs an invalid operation inside the `try` block:

```
1  # 11.py
2
3
4  class Operations:
5
6      @staticmethod
7      def divide(num, den):
8          # This method will raise an exception when denominator be 0
9          return float(num) / float(den)
10
11
12 try:
```

```
    # Here we manage the exceptions during the runtime of the function.
    # The first case will return an output and the second case will yield and
    # error beacuse denominator is 0. The output in this wont be printed.

    print('First case: {}'.format(Operations().divide(4, 5)))
    print('Second case: {}'.format(Operations().divide(4, 0)))

except ZeroDivisionError as err:
    print('Error: {}'.format(err))
```

```
First case: 0.8
Error: float division by zero
```

We can handle separately different types of exceptions by adding more specific exception blocks (ex: `ZeroDivisionError`, `TypeError`, `KeyError`, etc.), each one *catches* the exceptions according to the exception type that occurred:

```
# 12.py


class Operations:

    @staticmethod
    def divide(num, den):
        # Check if the input parameters are a valid type
        if not (isinstance(num, int) and isinstance(den, int)):
            raise TypeError('Invalid input type.')

        # Check the numerator and denominator are greater than 0
        if num < 0 or den < 0:
            # The message inside brackets will show once the
            # exception has been handled.
            raise Exception('Negative values. Check the input parameters')

        return float(num) / float(den)


```

```
# In this code section we manage the runtime exception using try and except
# sentences.

# First example, using float values
try:
    print('First case: {}'.format(Operations().divide(4.5, 3)))

except (ZeroDivisionError, TypeError) as err:
    # This block works with the already defined exception types
    print('Error: {}'.format(err))

except Exception as err:
    # This block only handles type Exception exceptions
    print('Error: {}'.format(err))


# Second example, using negative values
try:
    print('Second case: {}'.format(Operations().divide(-5, 3)))

except (ZeroDivisionError, TypeError) as err:
    # This block works with the already defined exception types
    print('Error: {}'.format(err))

except Exception as err:
    # This block only handles type Exception exceptions
    print('Error: {}'.format(err))
```

```
Error: Invalid input type.
Error: Negative values. Check the input parameters
```

If we do not use any particular exception name after the `Except` statement, it caught any exception triggered in the `try`. The `try` and `except` blocks can be complemented by the `else` and `finally` statements. The `else` block execute the instructions inside of it only in case no exception happened. The `finally` statement always executes the instructions defined in its block. This statement is commonly used to trigger cleaning actions, such as closing a file, database connections, etc. The following code shows an example:

```
# 13.py


class Operations:

    @staticmethod
    def divide(num, den):
        if not (isinstance(num, int) and isinstance(den, int)):
            raise TypeError('Invalid input type')

        if num < 0 or den < 0:
            raise Exception('Negative input values')

        return float(num) / float(den)


# The complete Try/Except structure

try:
    # Check if we can excecute this operation
    resultado = Operations.divide(10, 0)

except (ZeroDivisionError, TypeError):
    # This block works with the already defined exception types
    print('Check the input values. '
          'They aren\'t ints or the denominator is 0')

except Exception:
    # This block only handles type Exception exceptions
    print('All given values are negative')

else:
    # When we do no have errors, the program excecute this lines
    print('Everything is ok. They were no errors')

```

```
36 finally:
37     print('Rebember to ALWAYS use this structure to handle your runtime'
38           'errors')
```

```
Check the input values. They aren't ints or the denominator is 0
Rebember to ALWAYS use this structure to handle your runtimeerrors
```

5.4 Creating customized exceptions

In Python, there are three main types of exceptions: `SystemExit`, `KeyboardInterrupt` and `Exception`. All of the them inherits from `BaseException`. All the other exceptions suchs as the exceptions generated by errors iherits from the `Exceptions` class, as is shown in Figure 5.1:

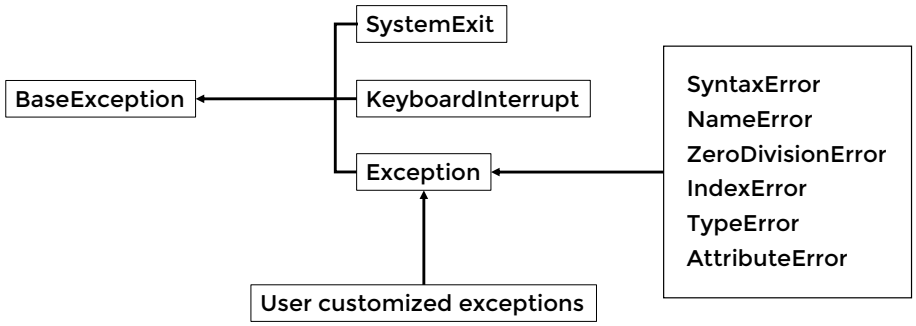


Figure 5.1: Diagram of exceptions hierarchy. All exceptions descended from a the general exception `BaseException`. We can create our new exceptions inheriting from the base class `Exception`.

The previous diagram explains the reason why using only the `Exception` statement without specifying exceptions catches any error. All of them are subclasses of `Exception`.

```
1 # 14.py
2
3
4 class Operations:
5
6     @staticmethod
7     def divide(num, den):
8         if not (isinstance(num, int) and isinstance(den, int)):
9             raise TypeError('Invalid input type')
```

```
10
11          if num < 0 or den < 0:
12              raise Exception('Negative input values')
13
14          return float(num) / float(den)
15
16
17  # In this section we handle the excetions
18  try:
19      print(Operations().divide(4, 0))
20
21  except Exception as err:
22      # This block works for all exception types.
23      print('Error: {}'.format(err))
24      print('Check the input')
```

```
Error: float division by zero
Check the input
```

To create customized exceptions, we need that the custom exception inherits from the `Exception` class. The following code shows an example:

```
1   # 15.py
2
3
4   class Exception1(Exception):
5       pass
6
7
8   class Exception2(Exception):
9
10      def __init__(self, a, b):
11          super().__init__("One of the values {0} or {1} is not integer"
12                           .format(a, b))
13
14
15  class Operations:
```

```
16
17      @staticmethod
18      def divide(num, den):
19          # In this example, we re-define exceptions that we used in the last
20          # examples.
21
22          if not (isinstance(num, int) and isinstance(den, int)):
23              raise Exception2(num, den)
24
25          if num < 0 or den < 0:
26              raise Exception1('Negative values\n')
27
28          return float(num) / float(den)
29
30
31  # This case raise the exception 1
32  try:
33      print(Operations().divide(4, -3))
34
35  except Exception1 as err:
36      # This block works for type one exception
37      print('Error: {}'.format(err))
38
39  except Excepcion2 as err:
40      # This block works for type two exception
41      print('Error: {}'.format(err))
42
43
44  # This case raise the exception 2
45  try:
46      print(Operations().divide(4.4, -3))
47
48  except Exception1 as err:
49      # This block works for type one exception
50      print('Error: {}'.format(err))
```

```
51
52 except Exception2 as err:
53     # This block works for type two exception
54     print('Error: {}'.format(err))
```

```
Error: Negative values

Error: One of the values 4.4 or -3 is not integer
```

Here we show another example:

```
1  # 16.py
2
3  class TransactionError(Exception):
4      def __init__(self, funds, expenses):
5          super().__init__("The money on your wallet is not enough to pay ${}"
6                           .format(expenses))
7          self.funds = funds
8          self.expenses = expenses
9
10     def excess(self):
11         return self.funds - self.expenses
12
13
14 class Wallet:
15     def __init__(self, money):
16         self.funds = money
17
18     def pay(self, expenses):
19         if self.funds - expenses < 0:
20             raise TransactionError(self.funds, expenses)
21         self.funds -= expenses
22
23 if __name__ == '__main__':
24     b = Wallet(1000)
25
26     try:
```

```
27          b.pay(1500)
28      except TransactionError as err:
29          print('Error: {}'.format(err))
30          print("There is an excess of expenses of ${}".format(err.excess()))
```

```
Error: The money on your wallet is not enough to pay $1500
There is an excess of expenses of $-500
```

Notes

Handling exceptions is another way of controlling the program's flow, similar to `if-else` sentences. It is recommended to use exceptions to control the errors in the program. We always can to create a sort of "error codes" for managing the returned values or results in different kind of operations. However, this solution makes the program and other modules hard to maintain. The number of error codes may grow at the same time the number of possible outputs we need to control. It makes our program impossible to be understood by any other programmers. A clearer example of the reason we need to handle exceptions is that in general, our program has to notify other applications that a particular error occurred. This kind of notifications would not be possible with the use of error codes. It is also critical that our code does not unexpectedly crash because during a crash the interpreter usually exposes in the output part of code that triggers the error. We have to avoid this situation if we correctly handle the exceptions.

5.5 Hands-On Activities

Activity 5

The Physics Club of your University *The Absolute Zeros* decided to program a calculator, whose main feature is that it handles letters and numbers as input. The person who was working on the project decided to change to the chemistry club *Cooler than Absolute Zero* and left the calculator unfinished. The calculator works well for some operations, but many times it triggers errors, and no one has been able to fix it. The president of the Physics Club has asked you to fix the code, such that it does not crash whenever is possible, handling the errors produced by user's input.

Consider that the president sent you a list of tested operations that work correctly: `tested_operations`; and a list with operations that do not work: `statements_for_testing`. We provide the file *AC05_0_provided_code.py* with the code. You cannot modify the code from line 87. Your job is to write the code to handle the exceptions and obtain the following output:

```
[ERROR] KeyError
```

```
Letter ‘g’ won’t be aggregated. It already exist in memory.
[ERROR] ZeroDivisionError
1 divided 0 is equal to infinite
[ERROR] KeyError
‘a’ doesn’t have any assigned values .It must be added before using it.
The operation a+2 wasn’t executed
9.81 plus 0 is equal to 9.81
88 divided by 2 is equal to 44.0
44.0 plus 0 is equal to 44.0
[ERROR] StopIteration
There’s one missing operator in 88/2+0+
[ERROR] ValueError
‘1=2’ cannot be parsed to float
[ERROR] The syntax ’1=2’ is incorrect. Read the manual for more information.
8.953 plus 1 is equal to 9.953
```

Chapter 6

Testing

A lot of programmers agree that "testing" is one of the most important aspects of the development of the software. Testing is the art of generating codes that can test our programs, checking that our development achieves the behavior requested by the final users. Even though we generally perform manual tests each time we develop a new piece of code that executes a task, it's very likely that we manually test for a rather typical case, which does not ensure that our new piece of code works in every possible scenario. Another factor to consider is efficiency: it takes a lot of time and code lines to set up and execute each evaluation. For these reasons, programmers prefer to automate testing since it quickly creates the set up for many different tests and allows to run them in a much visible way.

From now on, all your programs must go hand in hand with a program that tests it. On this chapter, we will see the fundamental concepts of testing and how to assemble unitary tests, but testing is a section that provides for an entire course.

In this chapter, we focus on unitary tests. A unitary test performs tests on minimal units of codes, such as functions or class methods. We present two Python libraries that make easier the creation of unitary tests: *unittest* y *Pytest*.

6.1 Unittest

The library `unittest` of Python gives many tools to create and run tests, one of the most important classes is "TestCase". In general, the classes that we create to perform tests must inherit the `TestCase` class. By convention all of the methods we implement to test must be called starting with the word "test", so that they are recognize at the moment of running the full program of testing in an automatic way:

```
1 import unittest
2
```

```
3
4  class CheckNumbers(unittest.TestCase):
5
6      # This test has to be ok
7      def test_int_float(self):
8          self.assertEquals(1, 1.0)
9
10     # This test fails
11     def test_str_float(self):
12         self.assertEquals(1, "1")
13
14 if __name__ == "__main__":
15     unittest.main()
```

```
.F
======================================================================
FAIL: test_str_float (__main__.CheckNumbers)
----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

The dot before the F indicates that the first test (test_int_float) passed with success. The F after the dot indicates that the second test failed. Afterward appear the details of the tests that failed, followed by the number of executed tests, the time it took and the total number of tests that failed.

We could then have all the tests we need inside the class that inherits TestCase, as long as the name of the method begins with “test”. Each test must be entirely independent of the others, in other words, the result of the calculus of a test must not affect the result of another test.

Assertion Methods

Assertion methods allow us to perform an essential kind of tests, where we know the desired result, and we just check if the value returned by the test matches that result. Assertion methods allow us to validate results in different ways. Some assertion methods (included in the class TestCase) are:

```python
import unittest


class ShowAsserts(unittest.TestCase):

    def test_assertions(self):
        a = 2
        b = a
        c = 1. + 1.
        self.assertEqual([1, 2, 3], [1, 2, 3])  # Fails if a != b
        self.assertNotEqual("hello", "bye")  # Fails if a == b
        self.assertTrue("Hello" == "Hello")  # Fails if bool(x) is False
        self.assertFalse("Hello" == "Bye")  # Fails if bool(x) is True
        self.assertIs(a, b)  # Fails if a is not b

        # Fails if a is b. Pay attention that "is" implies
        # equality (==) but not upside. Two differents objects
        # can have the same value.
        self.assertIsNot(a, c)

        self.assertIsNone(None)  # Fails if x is not None
        self.assertIsNotNone(2)  # Fails if x is None
        self.assertIn(2, [2, 3, 4])  # Fails if a is not in b
        self.assertNotIn(1, [2, 3, 4])  # Fails if a is in b

        # Fails if isinstance(a, b) is False
        self.assertIsInstance("Hello", str)
        # Fail if isinstance(a, b) is True
        self.assertNotIsInstance("1", int)

if __name__ == "__main__":
    unittest.main()
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

```
OK
```

The method `assertRaises` requires an exception, a function or any object with the implemented method `__call__` (*callable*) and an arbitrary number of arguments that will be passed to the *callable* method. The assertion will invoke the *callable* method with its arguments. It will fail if the method does not generate the expected error. The next code shows two ways of how to use the method `assertRaises`.

```
1  import unittest
2
3
4  def average(seq):
5      return sum(seq) / len(seq)
6
7
8  class TestAverage(unittest.TestCase):
9
10     def test_python30_zero(self):
11         self.assertRaises(ZeroDivisionError, average, [])
12
13     def test_python31_zero(self):
14         with self.assertRaises(ZeroDivisionError):
15             average([])
16
17 if __name__ == "__main__":
18     unittest.main()
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

In the second example `assertRaises` is used within a context manager (`with` sentence). It allows us to write our code in a more natural way by calling directly to the function `average` instead of having to call it indirectly as in the other example. We address context managers with more details in Chapter 10.

In Python 3.4 appeared new assertion methods:

- `assertGreater(first, second, msg=None)`
- `assertGreaterEqual(first, second, msg=None)`
- `assertLess(first, second, msg=None)`
- `assertLessEqual(first, second, msg=None)`
- `assertAlmostEqual(first, second, places=7, msg=None, delta=None)`
- `assertNotAlmostEqual(first, second, places=7, msg=None, delta=None)`

They test that `first` and `second` are approximately (or not approximately) equal, by calculating the difference, rounding the result number to `places` decimals places. If the argument `delta` is provided instead of "places", the difference between first and second must be less or equal (o more in case of `assertNotAlmostEqual`) than delta. If `delta` and `places` are given it generates an error.

The setUp method

Once we have written several tests, we realize we need to assemble a group of objects that will be used as input to compare the results of a test. The `setUp` method allows us to declare the variables that we will use and also takes care of resetting or re-initializing the variables before entering to a new test, in case that one of the other tests modified something in the variables.

```
1  from collections import defaultdict
2  import unittest
3
4
5  class StatisticList(list):
6
7      def mean(self):
8          return sum(self) / len(self)
9
10     def median(self):
11         if len(self) % 2:
12             return self[int(len(self) / 2)]
```

```
13          else:
14              idx = int(len(self) / 2)
15              return (self[idx] + self[idx-1]) / 2
16
17      def mode(self):
18          freqs = defaultdict(int)
19          for item in self:
20              freqs[item] += 1
21          mode_freq = max(freqs.values())
22          modes = []
23          for item, value in freqs.items():
24              if value == mode_freq:
25                  modes.append(item)
26          return modes
27
28
29  class TestStatistics(unittest.TestCase):
30
31      def setUp(self):
32          self.stats = StatisticList([1, 2, 2, 3, 3, 4])
33
34      def test_mean(self):
35          print(self.stats)
36          self.assertEqual(self.stats.mean(), 2.5)
37
38      def test_median(self):
39          self.assertEqual(self.stats.median(), 2.5)
40          self.stats.append(4)
41          self.assertEqual(self.stats.median(), 3)
42
43      def test_mode(self):
44          print(self.stats)
45          self.assertEqual(self.stats.mode(), [2, 3])
46          self.stats.remove(2)
47          self.assertEqual(self.stats.mode(), [3])
```

```
48
49 if __name__ == "__main__":
50     unittest.main()
```

```
[1, 2, 2, 3, 3, 4]
..[1, 2, 2, 3, 3, 4]
.
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Notice that the `setUp` method is never called explicitly inside of any of the tests because `unittest` does that for us. As you can see `test_median` modifies the list by adding a 4, but then in `test_mode` the list is the same as in the beginning. It occurs because `setUp` manages to re-initialize the variables that we need at the start of the each test. It helps us not to repeat code needlessly.

Besides the method `setUp`, `TestCase` offers us the method `tearDown`, which can be used to "clean" after all the tests have been executed. For instance, if our tests have the need to create some files, the idea is that at the end all of those temporary files are eliminated, in a way that ensures that the system is in the same state that it was before executing the tests:

```
1  import os
2  import unittest
3
4
5  class TestFiles(unittest.TestCase):
6
7      def setUp(self):
8          self.file = open("test_file.txt", 'w')
9          self.dictionary = {1: "Hello", 2: "Bye"}
10
11     def tearDown(self):
12         self.file.close()
13         print("Removing temporary files...")
14         os.remove("test_file.txt")
```

```
15
16      def test_str(self):
17          print("Writing temporary files...")
18          self.file.write(self.dictionary[1])
19          self.file.close()
20          self.file = open("test_file.txt", 'r')
21          d = self.file.readlines()[0]
22          print(d)
23          self.assertEqual(self.dictionary[1], d)
24
25
26  if __name__ == "__main__":
27      unittest.main()
```

```
Writing temporary files...
Hello
Removing temporary files...
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The discover module

When we test a program, we quickly begin to fill us with testing codes only. To solve this problem, we can arrange our modules that contain tests (objects TestCase) in more general modules called test suites (objects TestSuite), which include collections of tests:

```
1  import unittest
2
3
4  class ArithmeticTest(unittest.TestCase):
5
6      def test_arit(self):
7          self.assertEqual(1+1, 2)
8
```

```
9
10 if __name__ == "__main__":
11     Tsuite = unittest.TestSuite()
12     Tsuite.addTest(unittest.TestLoader().loadTestsFromTestCase(ArithmeticTest))
13     unittest.TextTestRunner().run(Tsuite)
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

How to ignore tests

Many times we know that some of the tests are going to fail in our program and we do not want they to fail during a given test. For example, when we have a function that is not yet finished. In these particular cases, we would like that the suite does not run these tests because we already know they will fail. Fortunately, *unittest* provides us with some decorators that mark tests and ignore them under certain circumstances. These decorators are: `expectedFailure()`, `skip(reason)`, `skipIf(condition, reason)`, `skipUnless(condition, reason)`:

```
1  import unittest
2  import sys
3
4
5  class IgnoreTests(unittest.TestCase):
6
7      @unittest.expectedFailure
8      def test_fail(self):
9          self.assertEqual(False, True)
10
11     @unittest.skip("Useless test")
12     def test_ignore(self):
13         self.assertEqual(False, True)
14
15     @unittest.skipIf(sys.version_info.minor == 5, "does not work on 3.5")
```

```
16      def test_ignore_if(self):
17          self.assertEqual(False, True)
18
19      @unittest.skipUnless(sys.platform.startswith("linux"),
20                           "does not work on linux")
21      def test_ignore_unless(self):
22          self.assertEqual(False, True)
23
24
25  if __name__ == "__main__":
26      unittest.main()
```

```
xsFs
======================================================================
FAIL: test_ignore_if (__main__.IgnoreTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "src/ENG/chapter_06/codes/unittest/6_ignore_test.py", line 17,
  in test_ignore_if self.assertEqual(False, True)
AssertionError: False != True

----------------------------------------------------------------------
Ran 4 tests in 0.001s
```

The `x` in the first line means *expected failure*, the `s` means *skipped test*, and the `F` means a real failure.

6.2 Pytest

Pytest is a framework for an alternative testing of *unittest*. It has a different design and allows to write the test in a more simple and readable way. *Pytest* also does not require that testing cases are classes, taking advantage of the fact that functions are objects, allowing any suitable function to work as a test. These features make the tests written in *Pytest* more readable and maintainable.

When we execute `py.test`, it begins by searching all the modules or sub-packages that begin with `test` in the current folder. Any function in that module that also begins with `test_` will be executed as an individual test. Whenever exists some class (inside the module) that starts with `Test`, any method inside that class beginning with

`test_` will also be executed in the testing environment. The next example shows how to write simple unitary tests; we can see that it is much shorter than it *unittest* counterparts:

```
1 def test_int_float():
2     assert 1 == 1.0
3
4
5 def test_int_str():
6     assert 1 == "1"
```

If we execute this code with `py.test PyTest0.py` in console the output is (assuming that in the file *PyTest0.py* is the code with the tests):

```
============================= test session starts ======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 2 items

PyTest0.py .F

================================== FAILURES ============================
_____________________________________ test_int_str _____________________________

    def test_int_str():
>       assert 1 == "1"
E       assert 1 == '1'

PyTest0.py:7: AssertionError
==================== 1 failed, 1 passed in 0.02 seconds ===============
```

We can see in the printed output some information about the platform. After that appears the name of the file that contains the tests; then we see the same notation used in *unittest* about the tests results, in this case anew the dot (".") indicates that the test has passed and the `F` that the test failed. We also see that it highlights the error and it mentions which kind of error it is.

We can also use classes in *pytest*. In this case, we do not have to inherit from any superclass from the testing module, as we did in *unittest*):

```
class TestNumbers:

    def test_int_float(self):
        assert 1 == 1.0

    def test_int_str(self):
        assert 1 == "1"
```

```
============================= test session starts ======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 2 items

PyTest1.py .F

================================= FAILURES =============================
_____________________________ TestNumbers.test_int_str ____________________

self = <PyTest1.TestNumbers object at 0x000001661A37EBE0>

    def test_int_str(self):
>       assert 1 == "1"
E       assert 1 == '1'

PyTest1.py:7: AssertionError
==================== 1 failed, 1 passed in 0.03 seconds ================
```

`setup` and `teardown` in Pytest

The main difference between the `setup` and `teardown` methods from *pytest* (more precisely `setup_method` and `teardown_method`) with the ones from *unittest*, is that in *pytest* these methods accept one argument: the function (object) that represents the method that is being called.

In addition, *pytest* provides the `setup_class` and `teardown_class` functions, that are methods that must test one class, they actually receive the class they are going to test as an argument

Finally, the `setup_module` and `teardown_module` methods are functions that run before and after all tests (despite they are functions or classes) in the module. The next example shows how to use each of these methods and in which order they are being executed:

```
1  def setup_module(module):
2      print("Setting up module {0}".format(module.__name__))
3
4
5  def teardown_module(module):
6      print("Tearing down module {0}".format(module.__name__))
7
8
9  def test_a_function():
10     print("Running test function")
11
12
13 class BaseTest:
14
15     def setup_class(cls):
16         print("Setting up Class {0}".format(cls.__name__))
17
18     def teardown_class(cls):
19         print("Tearing down Class {0}\n".format(cls.__name__))
20
21     def setup_method(self, method):
22         print("Setting up method {0}".format(method.__name__))
23
24     def teardown_method(self, method):
25         print("Tearing down  method {0}".format(method.__name__))
26
27
28 class TestClass1(BaseTest):
29
30     def test_method_1(self):
31         print("Running Method 1-1")
32
```

```
33      def test_method_2(self):
34          print("Running Method 2-1")
35
36
37  class TestClass2(BaseTest):
38
39      def test_method_1(self):
40          print("Running Method 1-2")
41
42      def test_method_2(self):
43          print("Running Method 2-2")
44
45  # Running by console "py.test PyTest.py -s",
46  # -s disables the deletion of "print" outputs.
```

```
============================= test session starts ======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: \codes\pytest, inifile:
collected 5 items

PyTest2.py Setting up module PyTest2
Running test function
.Setting up Class TestClass1
Setting up method test_method_1
Running Method 1-1
.Tearing down  method test_method_1
Setting up method test_method_2
Running Method 2-1
.Tearing down  method test_method_2
Tearing down Class TestClass1

Setting up Class TestClass2
Setting up method test_method_1
Running Method 1-2
.Tearing down  method test_method_1
Setting up method test_method_2
```

```
Running Method 2-2
.Tearing down  method test_method_2
Tearing down Class TestClass2

Tearing down module PyTest2


========================== 5 passed in 0.02 seconds ===================
```

Funcargs in pytest

A different way to do a setup and teardown in *pytest* is using *funcargs* (an abbreviation for function arguments). They correspond to variables that are previously set up in a file of tests configuration. It allows us to separate the configuration of the execution files from the tests files, and it also makes it possible to use the *funcargs* through multiple classes and modules.

To use the *funcargs*, we just add parameters to our testing functions, the names of these parameters will be used to search specific arguments in functions with a particular name (with the example this will become clearer). For instance, if we want to use the class `StatisticsList` in one of the previous examples by using *funcargs*:

```
1  def pytest_funcarg__valid_stats(request):
2      return StatisticsList([1, 2, 2, 3, 3, 4])
3
4
5  def test_mean(valid_stats):
6      assert valid_stats.mean() == 2.5
7
8
9  def test_median(valid_stats):
10     assert valid_stats.median() == 2.5
11     valid_stats.append(4)
12     assert valid_stats.median() == 3
13
14
15 def test_mode(valid_stats):
16     assert valid_stats.mode() == [2, 3]
```

```
17      valid_stats.remove(2)
18      assert valid_stats.mode() == [3]
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 3 items

PyTest3.py EEE

================================== ERRORS ============================
_____________________________ ERROR at setup of test_mean _____________________

request = <SubRequest 'valid_stats' for <Function 'test_mean'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined

PyTest3.py:3: NameError
_____________________________ ERROR at setup of test_median _________________

request = <SubRequest 'valid_stats' for <Function 'test_median'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined

PyTest3.py:3: NameError
_____________________________ ERROR at setup of test_mode _____________________

request = <SubRequest 'valid_stats' for <Function 'test_mode'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined
```

```
PyTest3.py:3: NameError
========================== 3 error in 0.05 seconds ==================
```

Note that the function has the prefix `pytest_funcarg__`. `valid_stats` contains the parameters used by the testing methods. In this case, `valid_stats` is returned by the `pytest_funcarg__valid_stats` function and corresponds to the list `StatisticsList([1,2,2,3,3,4])`. Note that in general the *funcargs* should be defined in an apart file called conftest.py, to allow the function be used by several modules. From the example we can see that the *funcargs* are reloaded every time for each test, hence we can modify the list inside a test as much as we need, without affecting other tests that may use the *funcargs* in further executions.

The `request.addfinalizer` method receives a function for cleaning purposes. It is equivalent to the `teardown` method, for example, it allows us to clean files, close connections, and empty lists. The next example tests the `os.mkdir` method (creates a directory) by checking that two temporary directories are indeed created:

```
1  import tempfile
2  import shutil
3  import os.path
4
5
6  def pytest_funcarg__temp_dir(request):
7      dir = tempfile.mkdtemp()
8      print(dir)
9
10     def cleanup():
11         shutil.rmtree(dir)
12
13     request.addfinalizer(cleanup)
14     return dir
15
16
17 def test_osfiles(temp_dir):
18     os.mkdir(os.path.join(temp_dir, 'a'))
19     os.mkdir(os.path.join(temp_dir, 'b'))
20     dir_contents = os.listdir(temp_dir)
21     assert len(dir_contents) == 2
```

```
22      assert 'a' in dir_contents
23      assert 'b' in dir_contents
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest4.py .

========================== 1 passed in 0.02 seconds =================
```

Skip tests in pytest

As in *unittest*, in *pytest* we can ignore some tests by using the `py.test.skip` function. It accepts only one argument; a string describing why we are ignoring the test. We can call `py.test.skip` anywhere. If we call it inside a test function, the test will be ignored; if we call it inside a module, all the tests within the module will be ignored, and if we call it inside a *funcarg* function, all the tests that call that *funcarg* will be ignored:

```
1  import sys
2  import py.test
3
4
5  def test_simple_skip():
6      if sys.platform != "Linows":
7          py.test.skip("This test only works on Linows OS")
8      fakeos.do_something_fake()
9      assert fakeos.did_not_happen
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest5.py s

========================== 1 skipped in 0.02 seconds ================
```

Note that we can call the function that ignores the tests inside of `if` sentences, which gives us a lot of conditioning possibilities. Besides the `skip` function, *pytest* offers us a decorator that allows us to skip tests when a particular condition is fulfilled. The decorator is `@py.test.mark.skipif(string)`, where the argument it receives corresponds to a string that contains a code that returns a boolean value after its execution.

```
1 import sys
2 import py.test
3
4
5 @py.test.mark.skipif("sys.platform != \"Linows\"")
6 def test_simple_skip():
7     fakeos.do_something_fake()
8     assert fakeos.did_not_happen
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest6.py s

========================== 1 skipped in 0.01 seconds =================
```

In *pytest*, there is another way to obtain a setup/teardown framework. This way is called *fixtures*. We will not address *fixtures* because we believe that it is out of the scope of this book, but you can find details about *fixtures* in this link: `https://pytest.org/latest/fixture.html\#fixture`.

6.3 Hands-On Activities

Activity 6.1

A management system has lately suffered attacks to its servers. The managers decided to increase their security by adding encryption to the information. You must test the encryption system.

The only things you know about the system are the following:

- The encrypter is symmetrical, which means, to encrypt and decrypt information, we use the same function. This is described in the `Encoder` class. The method to encrypt is called `encrypt`.

- The encrypter is based in two classes, called `Rotor` and `Reflector`.

- An *alphabet* is a set of allowed characters, it must be added before the encryption begins. To test the program, you can assume that the alphabet is lowercase ASCII. You must perform a test setup by calling the method `create_alphabet`, as shown in `main`.

- If we enter a word with characters that are not included in the alphabet, the program will raise a `ValueError` exception.

Rotor method

It takes a text archive and creates a function from it. It is not necessary that you know how it works, you only have to assert the properties of the function.

The application of this function is done through the `get` method. If the values are not in the domain, it returns `None`. To work correctly, it is necessary that this function is bijective.

Reflector method

It takes a text archive. The reflector is equal to the rotor, but it has a symmetry function. If $f(x) = y$ then $f(y) = x$. You access the function in the class through the method `get`. If the value is not in the domain, it returns `None`. It must show that bijection and symmetry hold. To test symmetry is enough to see if `get(x)`=y with `y` $\neq$ `None`. Also `get(y)`=`x` must hold.

Encrypter method

It has as initialization parameters a list of the rotors' addresses and the reflector's address.

To test that the encryption is correct you must prove for the following list of strings that by calling `encrypt`, the word gets encrypted and by calling it again, the word becomes decrypted.

`list_ = ['thequickbrownfoxjumpsoverthelazydog', 'python', 'bang', 'dragonfly', 'csharp']`.

To test the exception is enough to try with any word that has characters that are not in the alphabet. For instance, you may attempt with `'ñandu'`.

Notes

- Your code can be based in `encoder_test.py`.
- Check the `main` method if you have doubts on how to use the encrypter.
- To test the bijection of the functions, you can call the function `check_bijection` of the base code that receives a rotor or a reflector, and prove it holds bijection.

Summarizing, you have to perform the following steps:

- Initialize the alphabet as ASCII lowercase with a setup module.
- Prove that the rotor function is bijective. You must prove this for the archives (`rotor1.txt, rotor2.txt, rotor3.txt`)
- Prove that the reflector function is bijective and symmetric. You must prove this for the archive (`reflector.txt`)
- Prove that the encryption and decryption is correct. This includes to test that it raises the exception when needed.

Activity 6.2

The Bank has recently acquired last technology ATMs. The ATMs need to be programmed, for this reason, the bank has contacted you. You must assure that the coded features work correctly.

The features that come already implemented and you must test are:

- **Withdraw cash**: The user chooses an amount and withdraws it from its account. To obtain the money, the user must enter his password correctly. The maximum amount per transaction is $200.
- **Transfer money to a third party**: The user can transfer money to other users. The maximum amount to be transferred per operation is $1000.

For the withdraw money feature, create the following tests with **unittest**:

- Check that the credentials are correct.
- Check that it only withdraws money if there is enough balance.

- Check that once the money has been withdrawn, the amount is updated.

For the transfer money feature, create the following tests with **unittest**:

- Check that the account of the third person exists.
- Check that once the money has been transferred, both accounts have been updated with the correct amount.
- Check that if there is any error, the transference did not occur.

Chapter 7

Threading

7.1 Threading

Threads are the smallest program units that an operating system can execute. Programming with threads allows that several lightweight processes can run simultaneously inside the same program. Threads that are in the same process share the memory and the state of the variables of the process. This shared use of resources enables threads to run faster than execute several instances of the same program.

Each process has at least one thread that corresponds to its execution. When a process creates several threads, it executes these units as *parallel processes*. In a single-core machine, the parallelism is approximated through thread *scheduling* or *time slicing*. The approximation consists of assigning a limited amount of time to each thread repeatedly. The alternation between threads simulates parallelism. Although there is no true increase in execution speed, the program becomes much more responsive. For example, several tasks may execute while a program is waiting for a user input. Multi-core machines achieve a truly faster execution of the program. Figure 7.1 shows how threads interact with the main process.

Some examples of where it is useful to implement threads, even on single-core computers, are:

- Interfaces that interact with the user while the machine executes a heavyweight calculation process.
- Delegation of tasks that follow consumer-producer pattern, *i.e.*, jobs which outputs and inputs are related, but run independently.
- Multi-users applications, in which each thread would be in charge of the requests of each user.

Python 3 handles threads by using the *threading* library. It includes several methods and objects to manipulate threads.

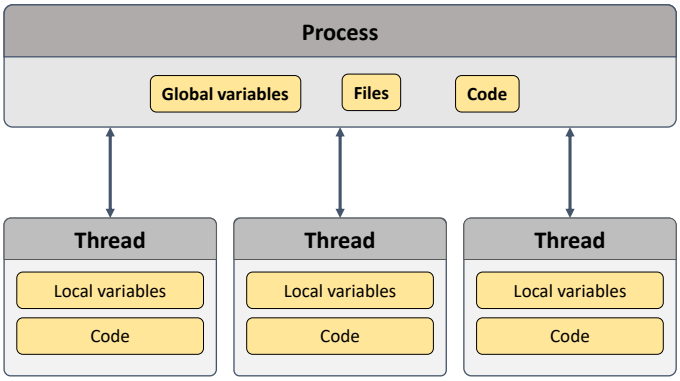


Figure 7.1: Diagram of a threading-based application

.

Creating Threads

We can create a new thread using the `Thread` class from the `Threading` library. This class requires three arguments: `target` to define the function to be executed; `name` to provide name we want to give to the thread; `args` to pass the `target` arguments. Once created, it may be executed by calling the `start()` method. In the next example, we create three threads `t1`, `w1`, and `w2`, that execute different instances of the `service` and `worker` functions.

```
1  # code0.py
2
3  import threading
4  import time
5
6
7  def worker():
8      print("{} starting...".format(threading.currentThread().getName()))
9      # This stops the thread execution for 2 seconds.
10     time.sleep(2)
11     print("{} exiting...".format(threading.currentThread().getName()))
12
13
14 def service():
15     print("{} starting...".format(threading.currentThread().getName()))
16     # This stops the thread execution for 4 seconds.
17     time.sleep(4)
18     print("{} exiting...".format(threading.currentThread().getName()))
```

```
19
20
21  # We create two named threads
22  t1 = threading.Thread(name='Thread 1', target=service)
23  w1 = threading.Thread(name='Thread 2', target=worker)
24
25  # This uses the default name (Thread-i)
26  w2 = threading.Thread(target=worker)
27
28  # All threads are executed
29  w1.start()
30  w2.start()
31  t1.start()
32
33
34  # The following will be printed before the threads finish executing
35  print('\nThree threads were created\n')
```

```
Thread 2 starting...
Thread-1 starting...
Thread 1 starting...

Three threads were created

Thread 2 exiting...
Thread-1 exiting...
Thread 1 exiting...
```

In the example, we see that once we have initialized the threads, the main program continues with the rest of the instructions while threads execute their task. The three threads end independently at different times. The main program waits until all the threads finish correctly.

The following code shows an example of how to pass arguments to the `target` function through the `args` attribute.

```
1  # code1.py
2
3  import threading
```

```
import time


def worker(t):
    print("{} starting...".format(threading.currentThread().getName()))

    # Thread is stopped for t seconds
    time.sleep(t)
    print("{} exiting...".format(threading.currentThread().getName()))


# Threads are created using the Thread class, these are associated with the
# objective function to be executed by the thread. Function attributes are
# given using the 'args' keyword. In this example, we only need to give one
# argument. For this reason a one value tuple is given.

w = threading.Thread(name='Thread 2', target=worker, args=(3,))
w.start()
```

```
Thread 2 starting...
Thread 2 exiting...
```

Another way of creating a thread is by inheriting from `Thread` and redefining the `run()` method.

```
# code2.py

import threading
import time


class Worker(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
```

```
14          print("{} starting...".format(threading.currentThread().getName()))
15          time.sleep(self.t)
16          print("{} exiting...".format(threading.currentThread().getName()))
17
18
19  class Service(threading.Thread):
20
21      def __init__(self, t):
22          super().__init__()
23          self.t = t
24
25      def run(self):
26          print("{} starting...".format(threading.currentThread().getName()))
27          time.sleep(self.t)
28          print("{} exiting...".format(threading.currentThread().getName()))
29
30
31  # Creating threads
32  t1 = Service(5)
33  w1 = Worker(2)
34  w2 = Worker(4)
35
36  # The created threads are executed
37  t1.start()
38  w1.start()
39  w2.start()
```

```
Thread-1 starting...
Thread-2 starting...
Thread-3 starting...
Thread-2 exiting...
Thread-3 exiting...
Thread-1 exiting...
```

Join()

In certain situations, we would like to synchronize part of our main program with the outputs of the running threads. When we need the main program to wait that the execution of a thread or a group of threads finished, we must use the `join(< maximum-waiting-time >)` method after the thread starts. In this way, every time we use `join()` the main program will be blocked until the referenced threads finish correctly. If we do not define the maximum waiting time, the main program waits indefinitely until the referenced thread finishes. Figure 7.2 shows the execution of the program using `join()`.

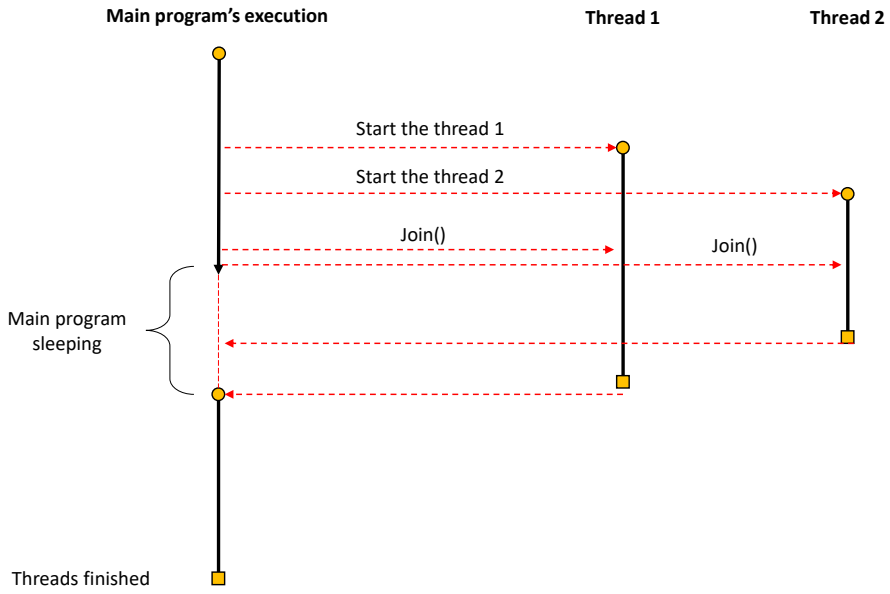


Figure 7.2: Diagram shows the program's flow when we use the `join()` method. We can see that the main program will sleep until *thread 1* finishes. The *thread 2* keeps running independently to the other thread and the main program.

Now let's see the same previous example but incorporating the `join()` method after threads start running.

```
1
2  # Creating threads
3  t1 = Service(5)
4  w1 = Worker(2)
5  w2 = Worker(4)
6
7  # Starting threads
8  t1.start()
9  w1.start()
```

```
10 w2.start()
11
12 # Here we call the join() method to block the main program.
13 # The other threads keep running independently
14 t0 = time.time()
15 w1.join()
16 print('Main program waits for: {}'.format(time.time() - t0))
```

```
Thread 1 starting...
Thread 2 starting...
Thread 3 starting...
Thread 2 exiting...
Main program waits for: 2.000131607055664
Thread 1 exiting...
Thread 3 exiting...
```

IsAlive()

We can identify if a thread finished its execution using the `IsAlive()` method or the `is_alive` attribute, for example, after using `join()`. The following example shows the way to use `IsAlive()` to check if a thread is still running after a certain amount of time.

```
1
2 t = Service(4)
3 t.start()
4
5 # The main program will wait 5 seconds after 't' has finished executing
6 # before continuing its execution.
7 t.join(5)
8
9 # This returns true if the thread is not currently executing
10 if not t.isAlive():
11     print('The thread has finished successfully')
12 else:
13     print('The thread is still executing')
```

```
Thread-1 starting...
```

```
Thread-1 exiting...
The thread has finished successfully
```

We can avoid the use of too many `prints` that help us with the tracking of threads, by using the *logging* library. Every time we make a log we have to embed the name of each thread on its log message, as shown in the following example:

```
1  # code5.py
2
3  import threading
4  import time
5  import logging
6
7
8  # This sets ups the format in which the messages will be logged on console
9  logging.basicConfig(level=logging.DEBUG, format='[%(levelname)s]'
10                                           '(%(threadName)-10s) %(message)s')
11
12 class Worker(threading.Thread):
13
14     def __init__(self, t):
15         super().__init__()
16         self.t = t
17
18     def run(self):
19         logging.debug('Starting')
20         time.sleep(self.t)
21         logging.debug('Exiting')
22
23
24 class Service(threading.Thread):
25
26     def __init__(self, t):
27         super().__init__()
28         self.t = t
29
```

```
30      def run(self):
31          logging.debug('Starting')
32          time.sleep(self.t)
33          logging.debug('Exiting')
34
35
36  # Creating threads
37  t1 = Service(4)
38  w1 = Worker(2)
39  w2 = Worker(2)
40
41  # Starting threads
42  w1.start()
43  w2.start()
44  t1.start()
```

```
[DEBUG](Thread-2  ) Starting
[DEBUG](Thread-3  ) Starting
[DEBUG](Thread-1  ) Starting
[DEBUG](Thread-2  ) Exiting
[DEBUG](Thread-3  ) Exiting
[DEBUG](Thread-1  ) Exiting
```

Daemon Threads

In general, the main program waits for all the threads to finish before ending its execution. *Daemon threads* let the main program to kill them off after other threads (and itself) finish. Daemon threads do not prevent the main program to end. In this way, daemon threads will only run until the main program finishes.

```
1  # code7.py
2
3  import threading
4  import time
5
6
7  class Worker(threading.Thread):
8
```

```
9       def __init__(self, t):
10          super().__init__()
11          self.t = t
12
13      def run(self):
14          print("{} starting...".format(threading.currentThread().getName()))
15          time.sleep(self.t)
16          print("{} exiting...".format(threading.currentThread().getName()))
17
18
19  class Service(threading.Thread):
20
21      def __init__(self, t):
22          super().__init__()
23          self.t = t
24          # We can set a thread as deamon inside the class definition
25          # setting the daemon attribute as True
26          self.daemon = True
27
28      def run(self):
29          print("{} starting...".format(threading.currentThread().getName()))
30          time.sleep(self.t)
31          print("{} exiting...".format(threading.currentThread().getName()))
32
33
34  # Creating threads
35  t1 = Service(5)
36  w1 = Worker(2)
37
38  # Setting the working thread as daemon
39  # We can use this same method when we define a function as target
40  # of a thread.
41  w1.setDaemon(True)
42
43  # Executing threads
```

```
44 w1.start()
45 t1.start()
```

```
Thread 2 starting...
Thread 1 starting...
```

The previous example explains the use of daemon threads. The console output shows how threads are interrupted abruptly after the main program ends its execution. We can compare this output with the output of the next example, configuring threads as daemon (removing lines 24 and 39):

```
Thread-2 starting...
Thread-1 starting...
Thread-2 exiting...
Thread-1 exiting...
```

Note that threads complete the execution and the program did not close until both threads finished. If for any reason, we require waiting for a daemon thread during an amount of time, we can specify that amount (in seconds) in the `join()` method:

```
1  # code9.py
2
3  import threading
4  import logging
5  import time
6
7  logging.basicConfig(level=logging.DEBUG,format='(%(threadName)-10s) '
8                                                  '%(message)s')
9
10 class DaemonThread(threading.Thread):
11
12     def __init__(self, t):
13         super().__init__()
14         self.t = t
15         self.daemon = True
16         self.name = 'daemon'
17
18     def run(self):
```

```
19          logging.debug('Starting')
20          time.sleep(self.t)
21          logging.debug('Exiting')
22
23  class NonDaemonThread(threading.Thread):
24
25      def __init__(self, t):
26          super().__init__()
27          self.t = t
28          self.name = 'non-daemon'
29
30      def run(self):
31          logging.debug('Starting')
32          time.sleep(self.t)
33          logging.debug('Exiting')
34
35  # Creating threads
36  d = DaemonThread(3)
37  t = NonDaemonThread(1)
38
39  # Executing threads
40  d.start()
41  t.start()
42
43  # Waiting thread d for 1 seconds
44  d.join(2)
45  print('is d alive?: {}'.format(d.isAlive()))
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
is d alive?: True
```

Timers

The class `Timer` is a subclass of the class `Thread` and allows us to execute a process or an action after a certain amount of time has passed. `Timer` requires as basic parameters the time in seconds after which the thread starts running, the name of the process to execute and the entry arguments for the process. The `cancel()` method allows us, if required, to cancel the execution of the timer before its begins.

```
1  # code10.py
2
3  import threading
4
5  def delayed_message(msg):
6      print("Message:", msg)
7
8  t1 = threading.Timer(10.0, delayed_message, args=("This is a thread t1!",))
9  t2 = threading.Timer(5.0, delayed_message, args=('This is a thread t2!',))
10 t3 = threading.Timer(15.0, delayed_message, args=('This is a thread t3!',))
11
12 # This thread will start after 10 seconds
13 t1.start()
14
15 # This thread will start after 5 seconds
16 t2.start()
17
18 # Here we cancel thread t1
19 t1.cancel()
20
21 # This thread will start after 15 seconds
22 t3.start()
```

```
Message: This is a thread t2!
Message: This is a thread t3!
```

7.2 Synchronization

Threads run in a non-deterministic way. Therefore, there are some situations in which more than one thread must share the access to certain resources, such as files and memory. During this process, **only one** thread have access to the

resource, and the remaining threads must wait for it. When there is multiple *concurrence* to a resource it is possible to control the access through synchronization mechanisms among the threads.

Locks

Locks allow us to synchronize the access to shared resources between two or more threads. The *Threading* library provides us with the `Lock` class which allows the synchronization. A lock has two states: *locked* and *unlocked*. The default state is *unlocked*. When a given thread t_i attempts to execute, first it tries to acquire the lock (with the `acquire()` method). If another thread t_j takes the lock, t_i must wait for t_j to finish and release it (with the `release()` method) to have the chance to acquire the lock. Once t_i acquires the lock, it can start executing. Figure 7.3 shows a general scheme of synchronization between threads using locks.

```
1  # code11.py
2
3  import threading
4
5
6  # This class models a thread that blocks to a file
7  class MyThread(threading.Thread):
8
9      lock = threading.Lock()
10
11     def __init__(self, i, file):
12         super().__init__()
13         self.i = i
14         self.file = file
15
16     # This method is the one executed when the start() method is called.
17     def run(self):
18
19         # Blocks other threads from entering the next block
20         MyThread.lock.acquire()
21         try:
22             self.file.write('This line was written by thread #{}\n'.format(self.i))
23         finally:
24             # Releases the resource
```

```
25              MyThread.lock.release()
26
27
28  if __name__ == '__main__':
29      n_threads = 15
30      threads = []
31
32      # We create a file to write the output.
33      with open('out.txt', 'w') as file:
34
35          # All writing threads are created at once
36          for i in range(n_threads):
37
38              my_thread = MyThread(i, file)
39
40              # The thread is started, which executes the run() method.
41              my_thread.start()
42              threads.append(my_thread)
```

```
This line was written by thread #0
This line was written by thread #1
This line was written by thread #2
This line was written by thread #3
This line was written by thread #4
This line was written by thread #5
This line was written by thread #6
This line was written by thread #7
This line was written by thread #8
This line was written by thread #9
This line was written by thread #10
This line was written by thread #11
This line was written by thread #12
This line was written by thread #13
This line was written by thread #14
```

Fortunately in Python *locks* can also work inside a *context manager* through the `with` sentence. In this case, is the

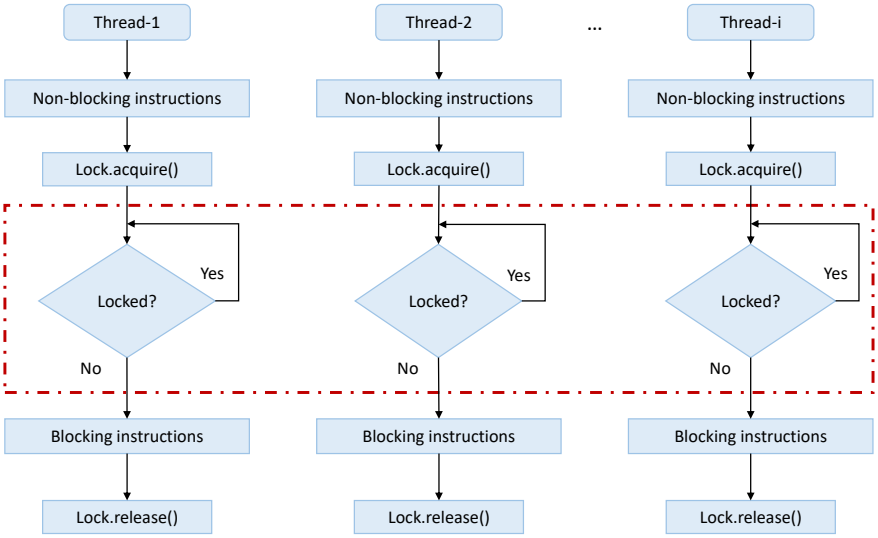


Figure 7.3: This image shows a general example where a set of *i* threads are running. Whenever a thread acquires the lock, the rest of the threads must wait to be able to execute. After the thread releases the lock, the other threads can acquire it and run their (blocking) instructions.

same `with` that is in charge of calling the `acquire()` and `release()` methods. For example, the *locks* used in the `run` method from the previous example can be implemented as shown:

```
1 def run(self):
2     with MyThread.lock:
3         self.file.write(
4             'This line was written by thread #{}\n'.format(self.i))
```

A common problem in concurrent programming is the *Producer-Consumer* pattern. This problem arises when two or more threads, known as *producers* and *consumers*, access to the same storage space or *buffer*. Under this scheme, producers put items in the *buffer* and consumers pull items out of the *buffer*. This model allows the communication between different threads. In general the *buffer* shared in this model is implemented through a *synchronized queue* or *secure queue*.

For example, let's assume that we can separate a program that processes a text file with numbers in two independent threads. The first thread is in charge of reading the file and appending the values to a queue. The second thread stores into another file the sum of the tuples of numbers previously added into the queue. We communicate both threads through a synchronized queue implemented as shown next:

```
# code13.py

import collections
import threading

class MyDeque(collections.deque):

    # We inherit from a normal collections module Deque and
    # we add the locking mechanisms to ensure thread
    # synchronization

    def __init__(self):
        super().__init__()
        # A lock is created for this queue
        self.lock = threading.Lock()

    def append(self, element):

        # The lock is used within a context manager
        with self.lock:
            super().append(element)
            print('[ADD] queue now has {} elements'.format(len(self)))

    def popleft(self):
        with self.lock:
            print('[REMOVE] queue now has {} elements'.format(len(self)))
            return super().popleft()
```

Now let's see the rest of the implementation of the producer and the consumer. As a recommendation, we encourage the read to try the examples directly in a terminal or using a IDE such as *PyCharm*.

```
# code14.py

import time
import threading

```

```
6
7  class Producer(threading.Thread):
8      # This thread is implemented as a class
9
10     def __init__(self, queue):
11         super().__init__()
12         self.queue = queue
13
14     def run(self):
15         # We open the file using a context manager. We explain this in details
16         # in Chapter 10.
17         with open('raw_numbers.txt') as file:
18             for line in file:
19                 values = tuple(map(int, line.strip().split(',')))
20                 self.queue.append(values)
21
22
23 def consumer(queue):
24     # This thread is implemented as a function
25
26     with open('processed_numbers.txt', 'w') as file:
27         while len(queue) > 0:
28             numbers = queue.pop()
29             file.write('{}\n'.format(sum(numbers)))
30
31             # Simulates that the consumer is slower than the producer
32             time.sleep(0.001)
33
34
35 if __name__ == '__main__':
36
37     queue = MyDeque()
38
39     p = Producer(queue)
40     p.start()
```

```
41
42      c = threading.Thread(target=consumer, args=(queue,))
43      c.start()
```

```
[ADD] queue now has 1 elements
[ADD] queue now has 2 elements
[ADD] queue now has 3 elements
[ADD] queue now has 4 elements
[ADD] queue now has 5 elements
```

Deadlock

In the context of multithreading-based applications, there is an innocent but dangerous situation in programs that use locks. This case is commonly called *deadlock*. A deadlock occurs when two or more threads are stuck waiting for each other to release a resource. For example, let `FirstProcess` be a thread that acquires a lock `a` and requests for a lock `b` so it can release the lock `a`. Let `SecondProcess` be another thread that already acquired the lock `b` and is waiting for the lock `a` before it releases the lock `b`. We note the deadlock because of our program will be frozen without getting a runtime error or crash. The next code example shows a template of a deadlock according to with the situation described:

```
1  # code15.py
2
3  import threading
4
5
6  class FirstProcess(threading.Thread):
7
8      def __init__(self, lock_a, lock_b):
9          self.lock_a = lock_a
10         self.lock_b = lock_b
11
12     def run(self):
13         with self.lock_a:
14             # Acquire the first lock
15
16             with self.lock_b:
```

```
17              # Acquire the second lock for another concurrent task
18
19
20  class SecondProcess(threading.Thread):
21
22      def __init__(self, lock_a, lock_b):
23          self.lock_a = lock_a
24          self.lock_b = lock_b
25
26      def run(self):
27          with self.lock_b:
28              # Acquire the first lock
29              # Notice that this thread require the lock_b, that
30              # could be taken fot other thread previously
31
32              with self.lock_a:
33                  # Acquire the second lock for another concurrent task
34
35
36  lock_a = threading.Lock()
37  lock_b = threading.Lock()
38
39  t1 = FirstProcess(lock_a, lock_b)
40  t2 = SecondProcess(lock_a, lock_b)
41  t1.start()
42  t2.start()
```

We can decrease the risk of a deadlock by restricting the number of locks that a threads can acquire at a time.

Queue

Fortunately, Python has an optimized library for secure queues management in *producer-consumer* models. The `queue` library has a implemented queue that safely manages multiples concurrences. It is different to the queue implemented in `collections` library used in data structures because that one does not have locks for synchronization.

The main queue methods in the `queue` library are:

- `put()`: Adds an item to the queue (push)
- `get()`: Removes and returns an item from the queue (pop)
- `task_done()`: Requires to be called each time an item has been processed
- `join()`: Blocks the queue until all the items have been processed

Recall the text file processing example shown before. The implementation using the `queue` library is as follows:

```
# code16.py

import threading
import time
import queue


class Producer(threading.Thread):

    def __init__(self, que):
        super().__init__()
        self.que = que

    def run(self):
        with open('raw_numbers.txt') as file:
            for line in file:
                values = tuple([int(l) for l in line.strip().split(',')])
                self.que.put(values)
                print('[PRODUCER] The queue has {} elements.'.format( \
                    self.que.qsize()))

                # Simulates a slower process
                time.sleep(0.001)


def consumer(que):
    with open('processed_numbers.txt', 'w') as file:
        while True:
```

```
29
30              # A try/except clause is used in order to stop
31              # the consumer once there is no elements left in the
32              # queue. If not for this, the consumer would be executing
33              # for ever
34
35              try:
36
37                  # If no elements are left in the queue, an Empty
38                  # exception is raised
39                  numbers = que.get(False)
40              except queue.Empty:
41                  break
42              else:
43                  file.write('{}\n'.format(sum(numbers)))
44                  que.task_done()
45
46                  # qsize() returns the queue size
47                  print('[CONSUMER] The queue now has {} elements.'.format( \
48                      que.qsize()))
49
50                  # Simulates a complex process. If the consumer was faster
51                  # than the producer, the threads would end abruptly
52                  time.sleep(0.005)
53
54
55  if __name__ == '__main__':
56
57      q = queue.Queue()
58
59      # a producer is created and executed
60      p = Producer(q)
61      p.start()
62
63      # a consumer thread is created and executed with the same queue
```

```
64      c = threading.Thread(target=consumer, args=(q,))
65      c.start()
```

```
[PRODUCER] The queue has 1 elements.
[CONSUMER] The queue now has 0 elements.
[PRODUCER] The queue has 1 elements.
[PRODUCER] The queue has 2 elements.
[PRODUCER] The queue has 3 elements.
[PRODUCER] The queue has 4 elements.
[CONSUMER] The queue now has 3 elements.
[CONSUMER] The queue now has 2 elements.
[CONSUMER] The queue now has 1 elements.
[CONSUMER] The queue now has 0 elements.
```

7.3 Hands-On Activities

Activity 7.1

ALERT! Godzilla has arrived at Santiago! Soldiers need to simulate a battle against Godzilla. The simulation will contribute deciding whether it is better to run away or to fight him. With this purpose, Soldiers have given us a report with the specifications that we have to accomplish. These specifications are:

- There is just one Godzilla and several soldiers.
- Each soldier has an attack speed, remaining life (HP), and strength of attack (damage).
- Godzilla attacks every eight seconds, affecting all soldiers by decreasing their HP in three units.
- Each time a soldier attacks Godzilla, it attacks back decreasing one-fourth of the soldiers' attack to his HP.
- The soldiers' attack speed is random between 4 and 19 seconds.
- You must create one new soldier every x seconds, where x has to be previously defined by you.

Activity 7.2

Congratulations! Thanks to the previous simulation (7.3), the Army has realized of its superiority against Godzilla. Santiago is safe again, or that is what we believe. The truth is that the epic battle has been nothing but a simulation

made by Godzilla to decide if he attacks Santiago or not. Now Santiago will be faced by Mega-Godzilla (Godzilla in its ultimate form), with all of the powers he has not shown before. The task is to simulate the battle between Mega-Godzilla and the Army so it can be written in history books. The simulation must:

- Contain several soldiers and one Mega-Godzilla.
- Each soldier has an attack speed, remaining life (HP), and strength of attack (damage).
- Mega-Godzilla attacks every N seconds, where N is a random number between 3 and 6. N has to be reset after each action.
- Mega-Godzilla attacks in the following ways:
  - Normal attack: Mega-Godzilla stomps affecting all soldiers. This attack causes a three units damage to each soldier.
  - Scaly Skin: Each time a soldier attacks Mega-Godzilla, it attacks back decreasing one-fourth of the soldiers' attack to his HP.
  - Ultimate Mega-Godzilla Super Attack: Mega-Godzilla screams causing a six units damage to every soldier. Also, the scream stuns all the soldiers for 10 seconds. During this period, soldiers can not perform any action.
- Soldiers' attack speed is random between 4 and 19 seconds.
- Soldiers' attack lasts a random number between 1 and 3 seconds.
- Only one soldier can attack Mega-Godzilla at a time.

Chapter 8

Simulation

During OOP modeling, we make assumptions on the system regarding the relationship between objects and data and use algorithms to represent their behavior. These models are just an approximation of real systems. Real systems include complex interactions usually hardly represented by exact analytical models. In these cases, systems' behavior must be simulated.

Simulations are used to generate data to obtain statistics of real systems. These statistics are used to make decisions on variables configuration that are relevant for systems' performance. For example, if we need to decide how many points of sale we must have in a supermarket, we can simulate customers arrival, products availability, shopping time, and also measure check out times. We can estimate the optimal number of points of sale that result in a desirable customers' check out time.

The main advantages of simulations are fast experimentation, cost, and risk reduction, design feedback, and data generation.

Simulations mainly depend on time and runtime. The former corresponds to the virtual clock that approximates the real time elapsed in the simulation. The latter represents the required computational time to carry out the simulation. In general, we want to simulate large amounts of time using a minimal amount of runtime.

Events occurrences are modeled using probability distributions to have more realistic simulations. For example, the arrival time of customers, or the customers' service time in a given store, can be modeled using an *exponential* distribution. For this kind of distribution, it is necessary to define the average rate of event occurrences. For instance, when a person arrives at a queue each 20 minutes, then this event has a distribution with a rate of 1/20. The following code shows an example of the use of the `expovariate` function to generate exponentially distributed times:

```
1 #00_expovariate.py
```

```
2
3  from random import expovariate
4
5  '''We added a basis time of 0.5 to prevent
6  time 0 returned by the distribution.'''
7
8  client_arrival_time = round(expovariate(1/20) + 0.5)
9  server_time_1 = round(expovariate(1/50) + 0.5)
10 server_time_2 = round(expovariate(1/50) + 0.5)
11
12 print(client_arrival_time)
13 print(server_time_1)
14 print(server_time_2)
```

```
29
53
26
```

8.1 Synchronous Simulation

It corresponds to one of the simpler ways of implementing a simulation. In this case, we divide the total simulation time into small intervals. At each interval, the program verifies all activities involved in the system. The general algorithm of this type of simulation is as follows:

while time simulation does not end **do**
    Increase the time by one unit
    if events occur in this time interval **then**
        Simulate events
    end if
end while

For instance, let's consider the case of modeling a car inspection station. This system operates as a queue, where the vehicles arrive randomly with probability P_c, and are processed by a station during a random amount of time. This type of problems is known as $M/M/k$ according to Kendal's notation. This notation defines that customers come to

the system in a *Markovian* way (M), the service time in the queue is also *Markovian* (M), and there are k servers to attend each car in the waiting queue.

```
# 01_synchronous.py

from collections import deque
import random


class Vehicle:
    """
    This class represent vehicles which arrives to the mechanical
    workshop
    """

    def __init__(self, vehicles):
        # When a new vehicle is created is chosen randomly incoming
        # vehicle type and the average time of service'''

        self.vehicle_type = random.choice(list(vehicles))
        self._review_time = round(
            random.expovariate(vehicles[self.vehicle_type]))

    @property
    def review_time(self):
        return self._review_time

    @review_time.setter
    def review_time(self, value):
        self._review_time = value

    def show_type(self):
        print("Being treated: {0} with an average time of {1} minutes"
              .format(self.vehicle_type, self.review_time))


```

```
34 class WorkShop:
35     """
36     This class represent the review line in the workshop.
37     """
38
39     def __init__(self):
40         self.current_task = None
41         self.review_time = 0
42
43     def busy(self):
44         return self.current_task is not None
45
46     def next_vehicle(self, vehicle):
47         self.current_task = vehicle
48         self.review_time = vehicle.review_time
49         vehicle.show_type()
50
51     def tick(self):
52         if self.current_task is not None:
53             self.review_time -= 1
54             if self.review_time <= 0:
55                 self.current_task = None
56
57
58 def new_vehicle_arrive():
59     """
60     This function returns if arrive a new vehicle to queue. It is
61     sampled from a uniform probability distribution. The method
62     returns True if the value delivered by the random function is
63     greater than a given value.
64     """
65     return random.random() >= 0.8
66
67
68 def technical_workshop(max_time, vehicles):
```

```
69      """
70       This function handles the process or technical service.
71      """
72
73      # Fix the random seed
74      random.seed(10)
75
76      # A WorkShop is created
77      workshop = WorkShop()
78
79      # Empty review line
80      review_line = deque()
81
82      # Waiting time
83      waiting_times = []
84
85      # The simulation cycle is defined until the maximum time in
86      # minutes, each time t is increased is evaluated if a new
87      # vehicle arrives at the review queue
88
89      for t in range(max_time):
90          if new_vehicle_arrive():
91              review_line.append(Vehicle(vehicles))
92
93          if not workshop.busy() and len(review_line) > 0:
94              # Next vehicle is taken out from review queue
95              curr_vehicle = review_line.popleft()
96              waiting_times.append(curr_vehicle.review_time)
97              workshop.next_vehicle(curr_vehicle)
98
99          # Decrease one tick of time to waiting vehicle
100         workshop.tick()
101
102     average_time = sum(waiting_times) / len(waiting_times)
103     total_time = sum(waiting_times)
```

```
104     print('Statistics:')
105     print('Average waiting time {0:6.2f} min.'.format(average_time))
106     print('Total workshop service time was', '{0:6.2f} min'.format(
107         total_time))
108     print('Total vehicles serviced: {0}'.format(len(waiting_times)))
109
110
111 if __name__ == '__main__':
112
113     # The types of vehicles and the average service time are defined
114     vehicles = {'motorcycle': 1.0 / 8, 'car': 1.0 / 15,
115                 'pickup_truck': 1.0 / 20}
116     maximum_time = 200
117     technical_workshop(maximum_time, vehicles)
```

Output:

```
Being treated: pickup_truck with an average time of 5 minutes
Being treated: car with an average time of 5 minutes
Being treated: motorcycle with an average time of 36 minutes
Being treated: motorcycle with an average time of 8 minutes
Being treated: motorcycle with an average time of 1 minutes
Being treated: car with an average time of 8 minutes
Being treated: pickup_truck with an average time of 6 minutes
Being treated: motorcycle with an average time of 9 minutes
Being treated: motorcycle with an average time of 14 minutes
Being treated: car with an average time of 2 minutes
Being treated: car with an average time of 43 minutes
Being treated: car with an average time of 9 minutes
Being treated: pickup_truck with an average time of 33 minutes
Being treated: car with an average time of 7 minutes
Being treated: pickup_truck with an average time of 20 minutes
Statistics:
Average waiting time  13.73 min.
Total workshop service time was 206.00 min
Total vehicles serviced: 15
```

Synchronous simulations require a lot of running time to produce results. Most of the time steps in the main simulation loop do not produce changes in the system. Verification of system's states and simulation constraints generates a waste of CPU time. Due to these downsides, in this chapter, we focus on *Discrete Event Simulation (DES)*.

8.2 Discrete Event Simulation (DES)

In DES paradigm exists a discrete sequence of events distributed in time, in which each event occurs at a determined instant t that generates a change in system's state. In contrast with the synchronous simulation, DES assumes that there are no variations in the system's states between consecutive events. This assumption allows us to jump directly to the next event, without wasting runtime. On each iteration, the simulation selects the next event by choosing the one that occurs first, according to its simulated time. The following pseudocode shows a general discrete-based event simulation algorithm:

while the events queue is not empty and the simulation time is not over **do**
    select the next event from the queue
    move the simulation time to the previously selected event's time
    simulate the event
end while

DES Model components

The following elements comprise a simulation model:

- A set of state variables that describe the system at any time. For example:

  A clock that stores the simulation time.

  A set of possible events, including the next instant they will take place.

- A set of simulation elements, for example:

  A method that controls the flow of different events.

  A set of performance variables, useful to keep simulation statistics.

Now we present an example of a technical car inspection station. Figure 8.1 shows the workflow of the system.

```
1 # 02_DES.py
2
```

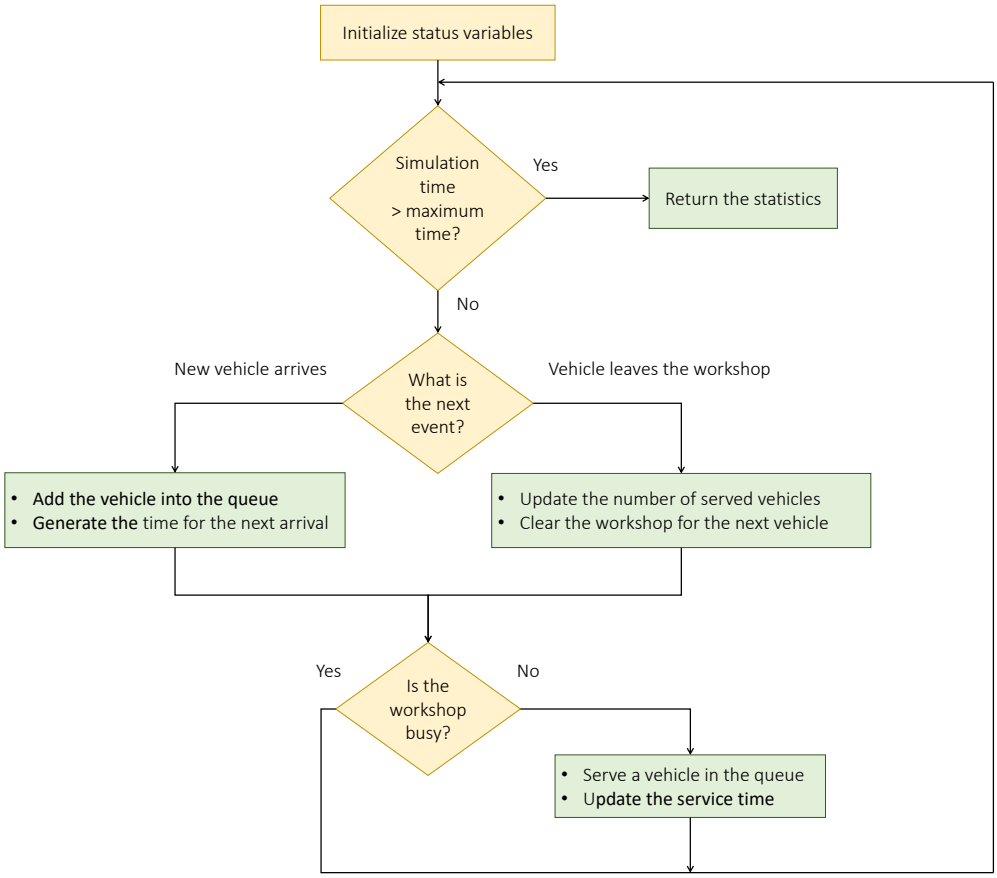


Figure 8.1: The figure shows a flow chart of the technical car inspection example. The gray rectangles describe the statistics updated on each event. The green boxes represent the simulation events. Decisions are represented by yellow diamonds.

```
3  from collections import deque
4  import random
5
6
7  class Vehicle:
8      """
9      This class represent vehicles which arrives to
10     the mechanical workshop
11     """
12
13     def __init__(self, arrival_time=0):
14         self.vehicle_type = random.choice(['motorcycle', 'pickup_truck', 'car'])
```

```
15          self.arrival_time = arrival_time
16
17      def __repr__(self):
18          return 'Vehicle type: {0}'.format(self.vehicle_type)
19
20
21  class WorkShop:
22      """
23      This class represents the workshop and its behaviors.
24      """
25
26      def __init__(self, types):
27          self.current_task = None
28          self.review_time = 0
29          self.types = types
30
31      def pass_vehicle(self, vehicle):
32          self.current_task = vehicle
33
34          # Create a random review time
35          current_type_rate = self.types[vehicle.vehicle_type]
36
37          # We add 0.5 to avoid random times equals to zero
38          self.review_time = round(random.expovariate(current_type_rate) + 0.5)
39
40      @property
41      def busy(self):
42          return self.current_task is not None
43
44
45  class Simulation:
46      """
47      This class implements the simulation.
48      Also you can use a function like in the previous case.
49      All variables used in the simulation are initialized.
```

```
50      """
51
52      def __init__(self, maximum_time, arrival_rate, types):
53          self.maximum_sim_time = maximum_time
54          self.arrival_rate = arrival_rate
55          self.simulation_time = 0
56          self.next_vehicle_time = 0
57          self.final_service_time = float('Inf')
58          self.waiting_time = 0
59          self.workshop = WorkShop(types)
60          self.waiting_line = deque()
61          self.served_vehicles = 0
62
63      def next_vehicle(self, arrival_rate):
64          # Update the arrival time of the next vehicle. We add 0.5 to avoid
65          # arrivals time equals to zero.
66          self.next_vehicle_time = self.simulation_time + \
67                                   round(random.expovariate(arrival_rate) + 0.5)
68
69      def run(self):
70          """
71          This method executes the simulation of the
72          workshop and the waiting line
73          """
74          random.seed(10)
75
76          self.next_vehicle(self.arrival_rate)
77
78          # The cycle is executed verified the simulation time is less than
79          # maximum simulation time
80          while self.simulation_time < self.maximum_sim_time:
81
82              # Update simulation time. Note that when the workshop is
83              self.simulation_time = min(self.next_vehicle_time,
84                                         self.final_service_time) if \
```

```
85                  self.workshop.busy else self.next_vehicle_time
86
87              print('[SIMULATION] time = {0} min'.format(self.simulation_time))
88
89              # First, review the next event between arrival and the final of a
90              # service
91              if self.simulation_time == self.next_vehicle_time:
92
93                  # If a vehicle has arrived we have to add it to the queue,
94                  # and to generate the next arrival.
95                  self.waiting_line.append(Vehicle(self.simulation_time))
96                  self.next_vehicle(self.arrival_rate)
97
98                  print('[QUEUE] {0} arrives in: {1} min.'.format(
99                      self.waiting_line[-1].vehicle_type,
100                     self.simulation_time))
101
102
103             elif self.simulation_time == self.final_service_time:
104
105                 print('[W_SHOP] Departure: {0} at {1} min.'.format(
106                     self.workshop.current_task.vehicle_type,
107                     self.simulation_time))
108
109                 self.workshop.current_task = None
110                 self.served_vehicles += 1
111
112             # If the workshop is busy, the vehicle has to wait for its turn,
113             # else can be served.
114
115             if not self.workshop.busy and len(self.waiting_line) > 0:
116                 # Get the next vehicle in the waiting line
117                 next_vehicle = self.waiting_line.popleft()
118
119                 # The vehicle begin to be served
```

```
120                 self.workshop.pass_vehicle(next_vehicle)
121
122                 # Update the waiting time, added 0 actually
123                 self.waiting_time += self.simulation_time \
124                                      - self.workshop.current_task.arrival_time
125
126                 # The next final time is generated
127                 self.final_service_time = self.simulation_time \
128                                           + self.workshop.review_time
129
130                 print('[W_SHOP] {0} enters with a expected service time {'
131                       '1} min.'.format(
132                     self.workshop.current_task.vehicle_type,
133                     self.workshop.review_time))
134
135         print('Statistics:')
136         print('Total service time {0} min.'.format(self.final_service_time))
137         print('Total number of served vehicles: {0}'
138               .format(self.served_vehicles))
139         w_time = self.waiting_time / self.served_vehicles
140         print('Average waiting time {0} min.'.format(round(w_time)))
141
142
143 if __name__ == '__main__':
144     # Set the arrival rate in 5 minutes.
145     arrival_rate_vehicles = 1 / 5
146
147     # Here we define different types of vehicles and the their service time.
148     vehicles = {'motorcycle': 1.0 / 8, 'car': 1.0 / 15,
149                 'pickup_truck': 1.0 / 20}
150
151     # The simulation runs until 50 minutes.
152     s = Simulation(70, arrival_rate_vehicles, vehicles)
153     s.run()
```

```
[SIMULATION] time = 5 min
```

```
[QUEUE] pickup_truck arrives in: 5 min.
[W_SHOP] pickup_truck enters with a expected service time 1 min.
[SIMULATION] time = 6 min
[W_SHOP] Departure: pickup_truck at 6 min.
[SIMULATION] time = 9 min
[QUEUE] pickup_truck arrives in: 9 min.
[W_SHOP] pickup_truck enters with a expected service time 35 min.
[SIMULATION] time = 18 min
[QUEUE] car arrives in: 18 min.
[SIMULATION] time = 27 min
[QUEUE] motorcycle arrives in: 27 min.
[SIMULATION] time = 31 min
[QUEUE] pickup_truck arrives in: 31 min.
[SIMULATION] time = 32 min
[QUEUE] car arrives in: 32 min.
[SIMULATION] time = 44 min
[W_SHOP] Departure: pickup_truck at 44 min.
[W_SHOP] car enters with a expected service time 1 min.
[SIMULATION] time = 45 min
[W_SHOP] Departure: car at 45 min.
[W_SHOP] motorcycle enters with a expected service time 16 min.
[SIMULATION] time = 61 min
[QUEUE] car arrives in: 61 min.
[SIMULATION] time = 61 min
[W_SHOP] Departure: motorcycle at 61 min.
[W_SHOP] pickup_truck enters with a expected service time 11 min.
[SIMULATION] time = 64 min
[QUEUE] car arrives in: 64 min.
[SIMULATION] time = 66 min
[QUEUE] motorcycle arrives in: 66 min.
[SIMULATION] time = 72 min
[QUEUE] car arrives in: 72 min.
Statistics:
Total service time 72 min.
Total number of served vehicles: 4
```

```
Average waiting time 18 min.
```

As we can see, the variation of the simulation time depends exclusively on the events that occur during the simulation. On each iteration, a vehicle gets into the waiting line, and it randomly generates the arrival time for the next car (or event). Then, each time a vehicle enters the station, it makes a change in the state of the system. In case the workshop is not busy the next car is served during a random service time. In the opposite case, while the station is busy the incoming vehicles accumulate in the queue. When a car leaves the workshop, the simulation time is updated, generating a new change in the system's state.

8.3 Hands-On Activities

Activity 8.1

The first branch office of **Seguritas Bank** has two tellers to attend all clients. Each teller has its queue. Clients arriving are placed in the shortest line, if both lines are equal, they prefer the teller 1. When a customer finishes his visit and leaves the cashier, the last client of the other queue checks if he can improve his position by changing between lines, in that case, he moves to the other line instantly. Assume that all clients arrive in a random time between one and three minutes. Also, each teller takes a random time between one and ten minutes to serve one client. You must use DES to simulate this situation during eighty minutes. Do not forget to identify the states variables and the relevant events.

Activity 8.2

The zoo *GoodZoo* is thinking about creating a new exhibition where they will show the life cycle in a natural environment as a simulation. Six species are interacting in the environment: Tiger, Elephants, Jaguars, Penguins, Grass and Cephalopods. Their interactions follow the rules of the food chain. They ask you to create the simulation with the following events:

- **Feeding rules** Animals should eat according to their `diet`. To eat, they have to wait a random time within a range defined in the variable `time_for_food`. After an animal selects a prey from the ecosystem, it verifies if the victim belongs to its diet. After eating, animals get `food_energy`. In case the prey is not included in the animal's diet, it loses half of `food_energy` and tries to eat in the next simulation time.

- **Birth** New animals born at a rate of `new_animal`. Parents lose an amount of energy defined in the variable `giving_birth_energy`. After the birth, you should verify that parents have enough energy to stay alive.

- **Deaths** An animal can die for three reasons: it has reached its `life_expectancy`, another animal has eaten it, and its total energy has got to zero.

The simulation parameters are in the table below. `food_frequency` is uniformly distributed within a range specified in the table. The time between births (`new_animal`) follows an exponential distribution where λ is also specified in the table:

Parameters	**Tiger**	**Elephant**	**Jaguar**	**Penguin**
`food`	Elephant, Jaguar, Penguin	Grass	Elephant, Tiger, Penguin	Cephalopod
`food_frequency`	$(20, 30)$	$(8, 15)$	$(35, 55)$	$(4, 15)$
`food_energy`	30	4	20	5
λ	$\frac{1}{75}$	$\frac{1}{200}$	$\frac{1}{80}$	$\frac{1}{80}$
`new_animal_energy`	15	7	10	10
`life_expectancy`	300	500	350	90
`initial number of animals`	5	8	5	12

The simulation never runs out of Grass and Cephalopods, they do not perform any action, and only Elephants and penguins eat them. Their quantities will **always** be:

- Grass: initial number of elephants $\times 3$
- Cephalopods: initial number of penguins $\times 5$

The maximum simulation time is 1000 units of time, and you must run 1000 of these simulations to calculate and show the following statistics:

1. Average simulation time.
2. How many times each species becomes extinct.
3. When does a species becomes extinct.
4. How many animals of each species were born.
5. How many animals of each species were eaten.
6. How many animals of each species ran out of energy.
7. How many animals of each species died old.
8. Average lifetime per species.
9. How many individuals of each species were alive at the end of the simulations.
10. How many animals of each species had to wait one turn or more to find their food.
11. Average food-waiting-time per species.

Requirements

- You must use Discrete Events Simulations because an iterative simulation won't be fast enough to test all the cases that *GoodZoo* wants to tests.
- Animals can be part of many events at the same time. The verification order of events it is not important, but you have to make sure that animals that die at event n cannot exist at $n + 1$.
- Animals become part of the simulation the next instant of time after they born. That means that if they are born at t, they cannot be food for any animal that eats at t, and they cannot die or find food. These actions can be performed only from $t + 1$.

Notes

- Consider making a detailed model; it will get your job much easier

Here we summarize all what you have to do:

- Simulate feeding
  - Randomly find prey. If the chosen animal cannot be eaten (it is not part of the diet) then wait for one instant.
  - Check energy level and simulate death if that corresponds
  - Calculate statistics 1, 2, 3, 8, 9, 10, 11 and the deaths-counts per species.
  - Compute the next feeding time event
- Simulate births
  - Add a new animal to the simulation
  - Simulate the energy loss and deaths if corresponds
  - Calculate statistic 4
  - Compute next birth time event
- Simulate deaths
  - Delete an animal from the ecosystem when it has reached its maximum age.
  - Calculate statistics 5, 6 and 7

Chapter 9

Handling Strings and Bytes

In this chapter, we present some of the most used methods in strings and bytes objects. Strings are extremely useful to manage most of the output generated from programs, like formatted documents or messages that involve program variables. Bytes allow us, among other uses, to perform input/output operations that make possible the communication between programs by sending/receiving data through different channels in low-level representation.

9.1 Some Built-in Methods for Strings

In Python, all strings are an immutable sequence of *Unicode* characters. *Unicode* is a standard encoding that allows us to have a virtual representation of any character. Here we have some different ways to create a string in Python:

```
1  a = "programming"
2  b = 'a lot'
3  c = '''a string
4  with multiple
5  lines'''
6  d = """Multiple lines with
7       double quotation marks """
8  e = "Three " "Strings" " Together"
9  f = "a string " + "concatenated"
10
11 print(a)
12 print(b)
13 print(c)
14 print(d)
```

```
15 print(e)
16 print(f)
```

```
programming
a lot
a string
with multiple
lines
Multiple lines with
     double quotation marks
Three Strings Together
a string concatenated
```

The type `str` has several methods to manipulate strings. Here we have the list:

```
1 print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Now we show some examples of methods to manipulate strings. We defer the reader to Python documentation for more examples of `string` methods.

The `isalpha()` method checks whether the string contains alpha characters or not. It returns `True` if all the characters are in the alphabet of some language:

```
1 print("abn".isalpha())
```

```
True
```

If there is a number, blank space or punctuation marks, it will return `False`:

```
print("t/".isalpha())
```

```
False
```

The `isdigit()` method returns `True` if all the characters in the string are digits:

```
print("34".isdigit())
```

```
True
```

We can check if a portion of a string includes a specific sub-sequence within it by `startswith()` and `endswith()` methods:

```
s = "I'm programming"
print(s.startswith("I'm"))
print(s.endswith("ing"))
```

```
True
True
```

If we require searching for a sub-sequence anywhere within a string we use the `find(seq)` method, which returns the index of *s* where the argument's sequence `seq` starts:

```
print(s.find("m p"))
```

```
2
```

The index method `index(str, beg=0 end=len(string)-1)` returns the index of where the sequence `str` starts within the string `s`. It always returns the first appearance of the argument `str` in `s` and starts at 0 as other Python indexing cases:

```
print(s.index('g'))
```

```
7
```

If we do not indicate the beginning or ending of the substring, `index()` method will use by default `beg=0` and `end=len(string)-1`. The next example shows how to search a substring that starts at position 4 and ends at position 10:

```
print(s.index('o', 4, 10))
```

```
6
```

Python will let us know if we use the right boundaries arguments to search:

```
print(s.index('i', 5, 10))
```

```
Traceback (most recent call last):
  File "2.py", line 29, in <module>
    print(s.index('i', 5, 10))
ValueError: substring not found
```

The `split()` method generates a list of words in `s` separated by blank spaces:

```
s = "Hi everyone, how are you?"
s2 = s.split()
print(s2)
```

```
['Hi', 'everyone,', 'how', 'are', 'you?']
```

By default `split()` uses blank spaces. The `join()` method let us to create a string concatenating the words in a list of strings through a specific character. The next example join the words in `s2` using the # character:

```
s3 = '#'.join(s2)
print(s3)
```

```
Hi#everyone,#how#are#you?
```

We can change portions of a string indicating the sub-sequence that we want to change and the character to replace:

```
print(s.replace(' ', '**'))
print(s)
```

```
Hi**everyone,**how**are**you?
```

The `partition(seq)` method splits any string at the first occurrence of the sub-sequence `seq`. It returns a tuple with the part before the sub-sequence, the sub-sequence and the remaining portion of the string:

```
s5 = s.partition(' ')
print(s5)
print(s)
```

```
('Hi', ' ', 'everyone, how are you?')
Hi everyone, how are you?
```

As we have seen in previous chapters, we can insert variable values into a string by using `format`:

```
# 4.py

name = 'John Smith'
grade = 4.5
if grade >= 5.0:
    result = 'passed'
else:
    result = 'failed'

template = "Hi {0}, you have {1} the exam. Your grade was {2}"
print(template.format(name, result, grade))
```

```
Hi John Smith, you have failed the exam. Your grade was 4.5
```

If we want to include braces within the string, we can escape them by using double braces. In the example below, we print a *Java* class definition:

```
# 5.py

template = """
public class {0}
{{
        public static void main(String[] args)
        {{
            System.out.println({1});
        }}
}}"""

print(template.format("MyClass", "'hello world'"));
```

```
public class MyClass
{
        public static void main(String[] args)
```

```
        {
            System.out.println('hello world');
        }
}
```

Sometimes we want to include several variables inside a string. This makes it hard to remember the order in which we have to write them inside of `format`. One solution is to use arguments with `keywords` in the function:

```
# 6.py

print("{} {label} {}".format("x", "y", label="z"))
```

```
x z y
```

```
# 7.py

template = """
From: <{from_email}>
To: <{to_email}>
Subject: {subject}
{message}
"""

print(template.format(
    from_email="someone@domain.com",
    to_email="anyone@example.com",
    message="\nThis is a test email.\n\nI hope this be helpful!",
    subject="This email is urgent")
)
```

```
From: <someone@domain.com>
To: <anyone@example.com>
Subject: This email is urgent

This is a test email.

I hope this be helpful!
```

We can also use lists, tuples or dictionaries as argument into `format`:

```
# 8.py

emails = ("a@example.com", "b@example.com")
message = {'subject': "You have an email.",
           'message': "This is an email to you."}

template = """
From: <{0[0]}>
To: <{0[1]}>
Subject: {message[subject]} {message[message]}
"""

print(template.format(emails, message=message))
```

```
From: <a@example.com>
To: <b@example.com>
Subject: You have an email. This is an email to you.
```

We can even use a dictionary and index it inside the string:

```
# 9.py

header = {"emails": ["me@example.com", "you@example.com"],
          "subject": "Look at this email."}

message = {"text": "Sorry this is not important."}

template = """
From: <{0[emails][0]}>
To: <{0[emails][1]}>
Subject: {0[subject]}
{1[text]}"""

print(template.format(header, message))
```

```
From: <me@example.com>
To: <you@example.com>
Subject: Look at this email.
Sorry this is not important.
```

We can also pass any object as an argument. For example, we can pass an instance of a class and then access to any of the attributes of the object:

```
# 10.py

class EMail:
    def __init__(self, from_addr, to_addr, subject, message):
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self.message = message

email = EMail("a@example.com", "b@example.com", "You have an email.",
              "\nThe message is useless.\n\nBye!")

template = """
From: <{0.from_addr}>
To: <{0.to_addr}>
Subject: {0.subject}
{0.message}"""
print(template.format(email))
```

```
From: <a@example.com>
To: <b@example.com>
Subject: You have an email.

The message is useless.

Bye!
```

We can also improve the format of the strings that we print. For instance, when you have to print a table with data, most of the time you want to show the values of the same variable aligned in columns:

```
1 # 11.py
2
3 items_bought = [('milk', 2, 120), ('bread', 3.5, 800), ('rice', 1.75, 960)]
4
5 print("PRODUCT  QUANTITY   PRICE   SUBTOTAL")
6 for product, price, quantity, in items_bought:
7     subtotal = price * quantity
8     print("{0:8s}{1: ^9d}    ${2: <8.2f}${3: >7.2f}"
9           .format(product, quantity, price, subtotal))
```

```
PRODUCT  QUANTITY   PRICE   SUBTOTAL
milk       120       $2.00    $ 240.00
bread      800       $3.50    $2800.00
rice       960       $1.75    $1680.00
```

Note that within each key there is a dictionary-type item, in other words, before the colon is the index of the argument within the `format` function. The string format is given after the colon. For example, `8s` means that the data is a string of 8 characters. By default, if the string is shorter than 8 characters, the rest is filled with spaces (on the right). If we enter a string longer than 8 characters, it will not be truncated. We can force longer strings to be truncated by adding a dot before the number that indicates the number of characters. As an example, if the format is `{1: ^9d}`:

- 1 corresponds to the index of the argument in the `format` function
- The space after the colon says that the empty spaces must be filled with spaces (by default integer types are filled with zeros)
- The symbol ^ is used to center the number in the available space
- `9d` means it will be an integer up to 9 digits

The order of these parameters, although they are optional, should be from left to right after the colon: character to fill the empty spaces, alignment, size and then type.

In the case of the price, `{2: <8.2f}` means that the used data is the third argument of the `format` function. The free places are filled with spaces; the < symbol means that the alignment is to the left, the number is a float of up to 8

characters, with two decimals. Similarly, in the case of the subtotal `{3:> 7.2f}`, it means that the used data is the fourth argument in the `format` function, the filling character is space, the alignment is to the right, and the number is a 7 digit float, including two decimal places.

9.2 Bytes and I/O

At the beginning of the chapter, we said that Python strings are an immutable collection of Unicode characters. Unicode is not a valid data storage format. We often read information of a string from some file or socket in bytes, not in Unicode. Bytes are the lowest level storage format. They represent a sequence of 8 bits which are described as an integer between 0 and 255, an equivalent hexadecimal between 0 and FF, or a literal (only ASCII characters are allowed to represent bytes).

Bytes can represent anything, such as coded character strings, or pixels of an image. In general, we need to know how they are coded to interpret the correct data type represented by bytes. For example, a binary pattern of 8 bits (one byte) may correspond to a particular character if is decoded as ASCII, or to a different character if is decoded as Unicode.

In Python, bytes are represented with the `bytes` object. To declare that an object is a byte you must add a *b* at the beginning of the object `a`. For example:

```
1  # 12.py
2
3  # What is between quotes is a byte object
4  characters = b'\x63\x6c\x69\x63\x68\xe9'
5  print(characters)
6  print(characters.decode("latin-1"))
7
8  # 61 and 62 are the hexadecimal representation of 'a' and 'b'
9  characters = b"\x61\x62"
10 print(characters.decode("ascii"))
11
12 # 97 and 98 are the corresponding ASCII code of 'a' and 'b'
13 characters = b"ab"
14 print(characters)
15 characters = bytes((97, 98))
16 print(characters)
17 print(characters.decode("ascii"))
```

```
b'clich\xe9'
cliché
ab
b'ab'
b'ab'
ab
```

```
# 13.py

# This generate an error because it is only possible to use ascii literals
# to create bytes

characters = b"áb"
```

```
------------------------------------------------------------------------------
SyntaxError Traceback (most recent call last)
<ipython-input-19-826203d742e2> in <module>()
---> 4 caracteres = b"áb"

SyntaxError: bytes can only contain ASCII literal characters.
```

The space symbol indicates that the two characters after the `x` correspond to a byte in hexadecimal digits. The bytes that coincide with the `ASCII` bytes are immediately recognized. When we print these characters, they appear correctly; the rest are printed as hexadecimal. The `b` reminds us that what is in the right is a `bytes` object, not a `string`. The sentence `characters.decode("latin-1")` decodes the sequence of bytes by using the `"latin-1"` alphabet.

The `decode` method returns a `Unicode` string. If we use another alphabet, we get another string:

```
# 14.py

characters = b'\x63\x6c\x69\x63\x68\xe9'
print(characters.decode("latin-1"))
print(characters.decode("iso8859-5"))
```

```
cliché
clich
```

To code a string in a different alphabet, we simply have to use the `encode` method from `str`. It is necessary to indicate the encoding alphabet:

```
1  # 15.py
2
3  characters = "estación"
4  print(characters.encode("UTF-8"))  # 8-bit Unicode Transformation Format
5  print(characters.encode("latin-1"))
6  print(characters.encode("CP437"))
7  print(characters.encode("ascii"))  # Can't enconde in ascii the character ó
```

```
b'estaci\xc3\xb3n'
b'estaci\xf3n'
b'estaci\xa2n'

---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-32-844dd5b7b18c> in <module>()
      3 print(characters.encode("latin-1"))
      4 print(characters.encode("CP437"))
----> 5 print(characters.encode("ascii")) # can't enconde in ascii the character ó

UnicodeEncodeError: 'ascii' codec can't encode character '\xf3' in position 6:
ordinal not in range(128)
```

The `encode` method has options to handle the cases where the string we want to code cannot be coded with the requested alphabet. These options are passed in key argument errors. Possible values are: `'strict'` by default to raise a `UnicodeDecodeError` exception, `'replace'` to replace an unknown character with symbol `?`, `'ignore'` to skip the character, and `'xmlcharrefreplace'` to create a xml entity that represents a Unicode character.

```
1  # 16.py
2
3  characters = "estación"
4  print(characters.encode("ascii", errors='replace'))
5  print(characters.encode("ascii", errors='ignore'))
```

```
6 print(characters.encode("ascii", errors='xmlcharrefreplace'))
```

```
b'estaci?n'
b'estacin'
b'estaci&#243;n'
```

In general, if we want to code a string and we do not know the alphabet we should use, the best to do is to use *UTF-8* because it is compatible with `ASCII`.

9.3 bytearrays

Just like the name suggests, *bytearrays* are arrays of bytes, that in contrast to bytes, they are mutable. They behave like lists: we can index them with slice notation, and we can add bytes to them with `extend`. To build a *bytearray* we must enter an initial byte:

```
1  # 17.py
2
3  ba_1 = bytearray(b"hello world")
4  print(ba_1)
5  print(ba_1[3:7])
6  ba_1[4:6] = b"\x15\xa3"
7  print(ba_1)
8  ba_1.extend(b"program")
9  print(ba_1)
10 # Here it prints an int, the ascii that corresponds to the letter 'h'
11 print(ba_1[0])
12 print(bin(ba_1[0]))
13 print(bin(ba_1[0])[2:].zfill(8))
```

```
bytearray(b'hello world')
bytearray(b'lo w')
bytearray(b'hell\x15\xa3world')
bytearray(b'hell\x15\xa3worldprogram')
104
0b1101000
01101000
```

Note that the last line is used to print the bits that correspond to the first byte. The `[2:]` is used to start from the third position, because the first two `b'` indicate that the format is binary. When we add `.zfill(8)` we mean that 8 bits will be used to represent a byte. It only makes sense when there are only zeros at the left side.

A one-byte character can be converted to an integer using the `ord` function:

```
1  # 18.py
2
3  print(ord(b"a"))
4  b = bytearray(b'abcdef')
5  b[3] = ord(b'g')  # The ASCII code for g is 103
6  b[4] = 68  # The ASCII code for D is 68, this is the same as b[4] = ord(b'D')
7  print(b)
```

```
97
bytearray(b'abcgDf')
```

9.4 Hands-On Activities

Activity 9.1

The teacher assistants of the *Advanced Programming* course, rescued at the last second all students data from the Computer Science Department's ERP. However, during the download some errors appeared and produced the following changes:

- In some rows, a random natural number followed by a blank space was placed at the beginning of the student's name. For example: *Jonh Smith* changed to *42 John Smith*. Those numbers must be removed.
- For each sequence of letters *n*, an additional *n* was added at the end of the concatenation. For example: *ANNE* became *ANNNE*. These names must be corrected. Assume that there are no names or last names with more than two consecutive *n*.
- Some names changed from uppercase to lowercase. All names must be in uppercase.

The rescued **students** list was saved as a **.csv** file. The main characteristic of this kind of files are:

- Each column is separated by commas

- The first row is the header for all columns.

You must convert all the data to three different formats: **LaTeX**, **HTML**, **Markdown**. The file *FORMATS.md* explains all the different formats and the representation that is required. Consider that the **students.csv** file always have three columns, but an unknown finite number of rows. Assume that there is no missing data. All the required files are available at `https://advancedpythonprogramming.github.io/`

Activity 9.2

One of your friends has to solve a big challenge: to separate a weird audio file that has two audio tracks mixed. Your friend did some research and found out that the file can be modified reading its data as bytes. The problem is that your friend has no idea about bytes, but he knows that you are reading this book, so he asked you to solve this big problem.

The input file, `music.wav` is a mix of two songs. You will have to read this file and generate two other files, `song1.wav` and `song2.wav`, each one with a separate audio. All the required files are available at `https://advancedpythonprogramming.github.io/`

The file `music.wav` has the mixed audio in the following way: From the byte 44 (start counting from 0), one byte corresponds to the first audio, and the other byte corresponds to the second. If `x` and `y` represent the bytes from the first and second audio respectively; the bytes are in the following order:

x y x y x y x y ...

Figure 9.1 shows the structure of a WAV audio file.

Consider the following:

- **The size of each file will change after the separation:** Each output file has half of the original data. Consider this when writing the headers of the output files. **Tip: Use the 4-byte representation for these integers.**
- The rest of the header data can be re-used *without modification* as they appear in the input, but do not forget to add them.

Bonus track

Now we have a new file, `audio.wav`. You suppose to do the same separation previously performed, but now the sampling frequency is different for each output file. In this case, bytes 25 to 28 and 29 to 32 are different depending on

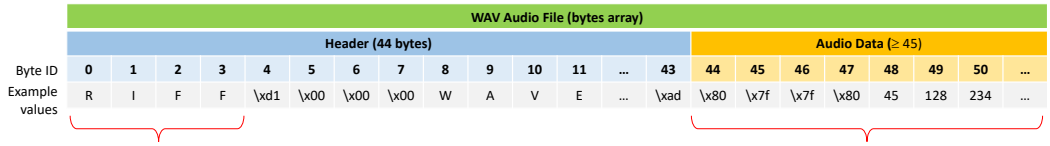
the file. The values are[1]:

- For the first file, write 11025 in both groups of bytes.
- For the second file, write 22050 in both groups of bytes.
- Run the previous algorithm with this audio and obtain the correct output.

Tips

- `int.from_bytes(bytes, byteorder='little')`: This function transform a byte to the equivalent integer.
- `int.to_bytes(size, byteorder='little')`: This function transforms an integer to bytes of the given size, with the given size of bytes.

[1] The sampling frequency corresponds to the number of samples in one unit of time from a continuous signal during the process of conversion from analogous to digital. The product is a discrete signal that can be reconstructed by a digital audio reproducer when it knows the sampling frequency.



Positions	Description
1-4	Defines the file as a "RIFF" file.
5-8	Size of the overall file minus 8 bytes (in bytes)
9-12	File type header. In this case "WAVE"
13-16	Format chunk marker "fmt"
17-20	Lenght of format data
21-22	Type of format (1 is PCM)
23-24	Number of Channels
25-28	Sample Rate (SR): number of samples per second
29-32	(SR * Bits per sample * Channels) / 8
33-34	(Bits per sample * Channels) / 8
35-36	Bits per sample
37-40	Data chunk header "data": marks the beginning of data section
41-44	Size of the data section
≥ 45	Audio data

Figure 9.1: Figure shows the WAV audio file structure. Note that Byte ID starts from 0 and the position in the structure form 1.

Chapter 10

I/O Files

So far we have worked reading and writing text files. However, operating systems represent most of its files as sequences of bytes, not as text. Since reading bytes and converting them to text is a very common operation in files, Python handles the bytes by transforming the string representation with the respective *encoders/decoders*. For example, the `open` function receive the name of the file to open, but also accept as an argument the character set for encoding the bytes, and the strategy to follow when bytes are inconsistent with the format. For instance, take a look at the different methods applied to the file *example_file*:

> *# Lorem Ipsum*
>
> *Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat.*

As we mentioned above, we open a file using the `open()` function:

```
1  # 19.py
2
3  file = open('example_file', 'r', encoding='ascii', errors='replace')
```

```
print(file.read())
file.close()

# Lorem Ipsum

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut
purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.
Curabitur dictum gravida mauris. Nam arcu libero, nonummy
eget, consectetuer id, vulputate a, magna. Donec vehicula augue
eu neque. Pellentesque habitant morbi tristique senectus et netus
et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra
metus rhoncus sem. Nulla et lectus vestibulum urna fringilla
ultrices. Phasellus eu tellus sit amet tortor gravida placerat.
```

We can override and overwrite this file, using the `'w'` argument in the `open()` method as follows:

```
# 20.py

content = "Sorry but now, this file will have me."
file = open('example_file', 'w', encoding='ascii', errors='replace')
file.write(content)
file.close()
```

Now, if we re-open and read the file like at the beginning:

```
# 21.py

file = open('example_file', 'r', encoding='ascii', errors='replace')
print(file.read())
file.close()

Sorry but now, this file will have me.
```

The contents in the file changed by the last sentence we wrote using the `'w'` argument. We could instead add content at the end of the file if we replace the `'w'` by an `'a'`:

```
# 22.py

content = "\nI will be added to the end."
```

```
file = open('example_file', 'a', encoding='ascii', errors='replace')
file.write(content)
file.close()

file = open('example_file', 'r', encoding='ascii', errors='replace')
print(file.read())
file.close()
```

```
Sorry but now, this file will have me.
I will be added to the end.
```

To open a file as binary, we only need to append the char `b` to the opening mode. For example, `'wb'` and `'rb'` instead of `'w'` and `'r'`, respectively. In this case, Python opens the file like text files, but without the automatic coding from byte to text:

```
# 23.py

content = b"abcde12"
file = open('example_file_2', 'wb')
file.write(content)
file.close()

file = open('example_file_2', 'rb')
print(file.read())
file.close()
```

```
b'abcde12'
```

We can concatenate bytes by simply using the `sum` operator. In the example below, we build dynamic content in each iteration. Then it is written in an explicit bytes file.

```
# 24.py

num_lines = 100

file = open('example_file_3', 'wb')
for i in range(num_lines):
    # To the bytes function we should pass an iterable with the content to
```

```
8       # convert. For this reason we pass the integer inside the list
9       content = b"line_" + bytes([i]) + b" abcde12"
10      file.write(content)
11  file.close()
```

To see the result, we re-read a fixed amount of bytes from the same file:

```
1  file = open('example_file_3', 'rb')
2  # The number 40 indicates the number of bytes that will be read from the file
3  print(file.read(40))
4  file.close()
```

```
b'line_\x00 abcde12line_\x01 abcde12line_\x02 abcde'
```

10.1 Context Manager

Every time we open a file, binary or not, we have to ensure our program close it correctly after reading the necessary information. However, exceptions may occur while the file is still open causing the loss of information and exposing a weakness in our code. One clear way is to close a file using the `finally` block, after a `try` statement. A cleaner option is to use a *context manager* which is responsible for executing the `try` and `finally` statements and manage the life-cycle of the object in the context without the need to write these statements directly. The following code shows an example of the using a *context*:

```
1  # 25.py
2
3  with open('example_file_4', 'r', errors='replace') as file:
4      content = file.read()
5  print(content)
```

```
file = open('example_file', 'r')
try:
    content = file.read()
finally:
    file.close()
```

If we execute `dir` in an `object` type, we can see that there are two methods called `__enter__` and `__exit__`:

```
1  # 26.py
```

```
2
3  file = open('example_file_4', 'w')
4  print(dir(file))
5  file.close()
```

```
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__iter__','__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
'__reduce_ex__','__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable',
'_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach',
'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode',
'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'seek',
'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
```

Both methods allow us to customize any object within a *context manager*. The `__exit__` method makes it possible to define the set of actions executed after a context finish. In the case of a file, it ensures that the context manager shall close the file correctly after reading the necessary data, even if an exception occurs while it is open.

In a similar way, the `__enter__` method let us specify the necessary steps performed to set the context of the object. For example, within a context the `open()` function returns a file object to the context manager. Finally, we simply use the `with` statement to generate the context and ensure that any object defined within it uses the `__enter__` and `__exit__` methods.

To personalize the use of any object within a *context manager*, we simply create a class and add the `__enter__` and `__exit__` methods. Then, call the class using the `with` statement. The following example shows how the `__exit__` method runs once we get out of the scope of the `with` statement.

```
1  # 27.py
2
3  import string
4  import random
5
6
7  class StringUpper(list):
8
```

```
9       def __enter__(self):
10          return self
11
12      def __exit__(self, type, value, tb):
13          for i in range(len(self)):
14              self[i] = self[i].upper()
15
16
17  with StringUpper() as s_upper:
18      for i in range(20):
19          # Here we randomly select a lower case ascii
20          s_upper.append(random.choice(string.ascii_lowercase))
21      print(s_upper)
22
23  print(s_upper)
```

```
['m', 'f', 'w', 'g', 'q', 'o', 'k', 'a', 'h', 'p', 'o', 'w', 'e', 'k', 'f',
't', 'e', 'n', 'm', 'l']
['M', 'F', 'W', 'G', 'Q', 'O', 'K', 'A', 'H', 'P', 'O', 'W', 'E', 'K', 'F',
'T', 'E', 'N', 'M', 'L']
```

In the last example we have a class that inherits from `list`. We implemented the `__enter__` and `__exit__` methods, hence we can instantiate it through a *context manager*. In this particular example, the *context manager* transforms all the lower case ascii characters to upper case.

10.2 Emulating files

We often have to interact with some software modules which only read and write data to and from files. In other cases, we just want to test a feature which requires some files. To avoid having to write data to persistent storage, we can have it on memory as files using `StringIO` or `BytesIO`. The next example shows the use of these modules:

```
1   # 28.py
2
3   from io import StringIO, BytesIO
4
5   # Simulate a text file
6   file_in = StringIO("info, text and more")
```

```
7  # Simulate a binary blob file
8  file_out = BytesIO()
9
10 char = file_in.read(1)
11 while char:
12     file_out.write(char.encode('ascii', 'ignore'))
13     char = file_in.read(1)
14
15 buffer_ = file_out.getvalue()
16 print(buffer_)
```

```
b'info, text and more'
```

Chapter 11

Serialization

The term *serialization* refers to the process of transforming any object into a sequence of bytes to be able to storage or transfer its data. We often use serialization to keep the results or states after a program finishes its execution. It may be very useful when another program or a later execution of the same program can load the saved objects and reuse them.

The Python `pickle` module allows us to serialize and deserialize objects. This module provides two principal methods

1. `dumps()` method: allows us to serialize an object.
2. `loads()` method: let us to deserialize the data and return the *original* object.

```
1 # 29.py
2
3 import pickle
4
5 tuple_ = ("a", 1, 3, "hi")
6 serial = pickle.dumps(tuple_)
7 print(serial)
8 print(type(serial))
9 print(pickle.loads(serial))
```

```
b'\x80\x03(X\x01\x00\x00\x00aq\x00K\x01K\x03X\x02\x00\x00\x00hiq\x01tq\x02.'
<class 'bytes'>
('a', 1, 3, 'hi')
```

Pickle has also the `dump()` and `load()` methods to serialize and deserialize through **files**. These methods are not the same methods `dumps()` and `loads()` described previously. The `dump()` method saves a file with the serialized object and the `load()` deserializes the content of the file. The following example shows how to use them:

```
# 30.py

import pickle

list_ = [1, 2, 3, 7, 8, 3]
with open("my_list", 'wb') as file:
    pickle.dump(list_, file)

with open("my_list", 'rb') as file:
    my_list = pickle.load(file)
    # This will generate an error if the object is not same we saved
    assert my_list == list_
```

The pickle module **is not safe**. You should never load a pickle file when you do not know its origin since it could run malicious code on your computer. We will not go into details on how to inject code via the pickle module, we refer the reader to [2] for more information about this topic. If we use Python 3 to serialize an object that will be deserialized later in Python 2, we have to pass an extra argument to `dump` or `dumps` functions, the argument name is `protocol` and must be equal to `2`. The default value is `3`). The next example shows how to change the pickle protocol:

```
# 31.py

import pickle

my_object = [1, 2, 3, 4]
serial = pickle.dumps(my_object, protocol=2)
```

When pickle is serializing an object, what is trying to do is to save the attribute `__dict__` of the object. Interestingly, before checking the attribute `__dict__`, pickle checks if there is a method called `__getstate__`, if any, it will serialize what the method `__getstate__` returns instead of the dictionary `__dict__` of the object. It allows us to customize the serialization:

```
# 32.py
```

```
2
3  import pickle
4
5
6  class Person:
7
8      def __init__(self, name, age):
9          self.name = name
10         self.age = age
11         self.message = "Nothing happens"
12
13     # Returns the current object state to be serialized by pickle
14     def __getstate__(self):
15         # Here we create a copy of the current dictionary, to modify the copy,
16         # not the original object
17         new = self.__dict__.copy()
18         new.update({"message": "I'm being serialized!!"})
19         return new
20
21 m = Person("Bob", 30)
22 print(m.message)
23 serial = pickle.dumps(m)
24 m2 = pickle.loads(serial)
25 print(m2.message)
26 print(m.message)  # The original object is "the same"
```

```
Nothing happens
I'm being serialized!!
Nothing happens
```

Naturally, we can also customize the serialization by implementing the `__setstate__` method, it will run each time you call `load` or `loads`, for setting the current state of the newly deserialized object. The `__setstate__` method receives as argument the state of the object that was serialized, which corresponds to the value returned by `__getstate__`. `__setstate__` must set the state in which we want the deserialized object to be by setting `self.__dict__`. For instance:

```
1  # 33.py
```

```
2
3  import pickle
4
5
6  class Person:
7
8      def __init__(self, name, age):
9          self.name = name
10         self.age = age
11         self.message = "Nothing happens"
12
13     # Returns the current object state to be serialized by pickle
14     def __getstate__(self):
15         # Here we create a copy of the current dictionary, to modify the copy,
16         #  not the original object
17         new = self.__dict__.copy()
18         new.update({"message": "I'm being serialized!!"})
19         return new
20
21     def __setstate__(self, state):
22         print("deserialized object, setting its state...\n")
23         state.update({"name": state["name"] + " deserialized"})
24         self.__dict__ = state
25
26 m = Person("Bob", 30)
27 print(m.name)
28 serial = pickle.dumps(m)
29 m2 = pickle.loads(serial)
30 print(m2.name)
```

```
Bob
deserialized object, setting its state...

Bob deseialized
```

A practical application of `__getstate__` and `__setstate__` methods can be when we need to serialize an

object that contains attributes that will lose sense after serialization, such as, a database connection. A possible solution is: first to use `__getstate_` to remove the database connection within the serialized object; and then manually reconnect the object during its deserialization, in the `__setstate__` method.

11.1 Serializing web objects with JSON

One disadvantage of pickle serialized objects is that only other Python programs can deserialize them. JavaScript Object Notation (**JSON**) is a standard data exchange format that can be interpreted by many different systems. JSON may also be easily read and understood by humans. The format in which information is stored is very similar to Python dictionaries. JSON can only serialize data (`int`, `str`, `floats`, `dictionaries` and `lists`), therefore, you can not serialize functions or classes. In Python there is a module that transforms data from Python to JSON format, called `json`, which provides an interface similar to `dump(s)` and `load(s)` in pickle. The output of a serialization using the `json` module's `dump` method is of course an object in JSON format. The following code shows an example:

```
1  # 34.py
2
3  import json
4
5
6  class Person:
7
8      def __init__(self, name, age, marital_status):
9          self.name = name
10         self.age = age
11         self.marital_status = marital_status
12         self.idn = next(Person.gen)
13
14     def get_id():
15         cont = 1
16         while True:
17             yield cont
18             cont += 1
19
20     gen = get_id()
21
22 p = Person("Bob", 35, "Single")
```

```
23 json_string = json.dumps(p.__dict__)
24 print("JSON data: ")
25 print(json_string)
26 print("Python data: ")
27 print(json.loads(json_string))
```

```
JSON data:
{"marital_status": "Single", "name": "Bob", "idn": 1, "age": 35}
Python data:
{'marital_status': 'Single', 'name': 'Bob', 'age': 35, 'idn': 1}
```

We can also write directly JSON objects as Python strings that follow the JSON data format. In the next instance we create a JSON object type directly (without `json.dumps`), and then we deserialize it into a Python type object with `json.loads`:

```
1 # 35.py
2
3 import json
4
5
6 json_string = '{"name":"Mark","age":34,' \
7               '"marital_status": "married", "score" : 90.5}'
8 print(json.loads(json_string))
```

```
{'marital_status': 'married', 'age': 34, 'score': 90.5, 'name': 'Mark'}
```

We can also load data with particular formats. For instance, in the case we want to show `int` types as `floats`:

```
1 # 36.py
2
3 import json
4
5
6 json_string = '{"name":"Mark","age":34,' \
7               '"marital_status": "married", "score" : 90.5}'
8 print(json.loads(json_string, parse_int=float))
```

```
{'age': 34.0, 'name': 'Mark', 'score': 90.5, 'marital_status': 'married'}
```

In Python, by default, JSON converts all data to a dictionary. If you want to turn data into another type, we can use the object argument `object_hook` with a `lambda` function that will be applied to each data object. For instance, if we want to load JSON data into a list of tuples instead of a dictionary:

```
# 37.py

import json


json_string = '{"name":"Mark","age":34,' \
              '"marital_status": "married", "score" : 90.5}'
data = json.loads(json_string,
                  object_hook=lambda dict_obj:
                  [tuple((i, j)) for i, j in dict_obj.items()])
print(data)
```

```
[('marital_status', 'married'), ('age', 34), ('name', 'Mark'), ('score', 90.5)]
```

We can create any function and then apply it to the data we want to convert:

```
# 38.py

import json


def funcion(dict_obj):
    collection = []
    for k in dict_obj:
        collection.extend([k, str(dict_obj[k])])
    return collection

json_string = '{"name":"Mark","age":34,' \
              '"marital_status": "married", "score" : 90.5}'
data = json.loads(json_string, object_hook=lambda dict_obj: funcion(dict_obj))
print(data)
```

```
['age', '34', 'name', 'Mark', 'score', '90.5', 'marital_status', 'married']
```

We can also customize the way we code the data in JSON format by creating a class that inherits from the `json.JSONEncoder` class and by overriding the default method:

```
# 39.py

import json
from datetime import datetime


class Person:
    def __init__(self, name, age, marital_status):
        self.name = name
        self.age = age
        self.marital_status = marital_status
        self.idn = next(Person.gen)

    def get_id():
        cont = 1
        while True:
            yield cont
            cont += 1

    gen = get_id()


class PersonaEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Person):
            return {'Person_id': obj.idn, 'name': obj.name,
                    'age': obj.age, 'marital_status': obj.marital_status,
                    'dob': datetime.now().year - obj.age}
        return super().default(obj)


p1 = Person("Bob", 37, "Single")
p2 = Person("Mark", 33, "Married")
```

```
34 p3 = Person("Peter", 24, "Single")
35
36 print("Default serialization:\n")
37 # With this we serialized using the default method
38 json_string = json.dumps(p1.__dict__)
39 print(json_string)
40
41 #  Now we serialize with the personalized method
42 print("\nCustom Serialization:\n")
43 json_string = json.dumps(p1, cls=PersonaEncoder)
44 print(json_string)
45 json_string = json.dumps(p2, cls=PersonaEncoder)
46 print(json_string)
47 json_string = json.dumps(p3, cls=PersonaEncoder)
48 print(json_string)
```

```
Default serialization:

{"name": "Bob", "marital_status": "Single", "age": 37, "idn": 1}

Custom Serialization:

{"age": 37, "name": "Bob", "marital_status": "Single", "Person_id": 1,
"dob": 1978}
{"age": 33, "name": "Mark", "marital_status": "Married", "Person_id": 2,
"dob": 1982}
{"age": 24, "name": "Peter", "marital_status": "Single", "Person_id": 3,
"dob": 1991}
```

11.2 Hands-On Activities

Activity 11.1

The *Walkcart* supermarket has asked us to help them to handle its clients' transactional data. This information, in future, will be useful to know who are the best customers and cashiers. The *Walkcart* managers asked us to implement

all the needed functionalities to allow the clerk to update and save clients' data. This information has to be stored in a file inside the *ClientsDB* folder. All files need to have the `.walkcart` extension.

Each cashier should be able to:

- Star a new session with their name. To verify that the person is really a cashier, you can verify the file in `cashiers.walkcart` that contains a list of serialized strings.
- For each client, ask its name `(str)`, **id** `(int)` and money spent `(int)`.
- Update and generate the file `id.walkcart` inside the `ClientsDB` folder, where `id` is the client's id. This file must be a serialization of the `Client` class, that you must define. You have to save: *name*, *id*, *accumulated spent*, and **last purchase's date**.

The company should also be able to:

- Star a session with a unique user: *WalkcartUnlimited*.
- Generate the file `TOP.walkcart`. This file must be in `format` and must contain the data of the client who historically has spent the most. If you find more than one winner, pick one randomly.

Notes

- If the username does not correspond to a cashier (or a *WalkcartUnlimited*), then this user must not be able to log into the system.
- At all times you must serialize using `pickle`.
- Each cashier can attend a customer more than once, and one client can make as many purchases as he/she want. Keep the information in the file correct and updated.
- The `PlainTextInfo.txt` file cannot be used by your program.

Chapter 12

Networking

Computer networks allow communication between multiple computers, regardless of their physical location. Internet provides us with an infrastructure that allows computers to interact across the entire web. In this chapter, we explore the main concepts needed to understand communication protocols and to learn how to send and receive data through networks with Python.

12.1 How to identify machines on internet

Every machine connected to internet has an IP (*Internet Protocol*) address. Every website has (besides the IP address) a *hostname*. For example, the *hostname* `www.python.org` has `199.27.76.223` as its IP address. We can obtain a website's IP address by typing `ping` *`hostname`* on a unix terminal. For example, type `ping www.python.org` in a unix or windows terminal.

An IP address on its fourth version (*IPv4*) corresponds to a binary number of 32 bits (grouped by 4 bytes), hence in the *IPv4* format we can have a maximum of $(2^8)^4 = 256^4 = 4.294.967.296$ IP addresses. Because the maximum number of IP addresses was surpassed by the amount of machines connected to the global network, the new *IPv6* standard was created. In *IPv6*, every address has 128 bits, divided in 8 groups of 16 bits each, represented in hexadecimal notation. For example: `20f1:0db8:0aab:12f1:0110:1bde:0bfd:0001`

12.2 Ports

An IP address is not enough to establish a connection. When two computers are communicating through an application, besides the IP address, we must specify a port. A port establishes the communication channel that the application uses inside the machine. The sender and receiver applications can have different ports assigned on their respective

machines, even if the applications are the same. For example, if we want to start communicating with a remote server through **FTP** (*File Transfer Protocol*), we must connect to the server by its IP address and port 21. There are many applications with already assigned ports, as we can see on Table 12.1.

Table 12.1: Some preset ports

Port	Description
21	FTP CONTROL
22	SSH
23	Telnet
25	SMTP (email)
37	Time
42	Host Name Server (Nameserv)
53	Domain Name System (DNS)
80	HTTP (Web)
110	POP3 (email)
118	SQL Services
119	NNTP (News)
443	HTTPS (Web)

The port number is represented by 16 bits, hence exist $2^{16} = 65536$ possible ports. There are three preset ranges in the list of available ports: *well-known ports* in the range $[0 - 1023]$, *registered ports* in the range $[1024 - 49151]$ and *dynamic or private ports* in the range $[49152 - 65535]$. The IANA (*Internet Assigned Numbers Authority*) organization is responsible for designing and maintaining the number of ports for the first two ranges. The third range, in general, is used by the operating system, that automatically assigns ports depending on the running programs' requests. Every running program must be represented by a host and a port to communicate inside a network. In Python, we represent those values as a tuple. For example: `("www.yahoo.es", 80)` or `("74.6.50.150", 443)`.

The most used transmission protocols in a network are: **TCP (*Transmission Control Protocol*)** y **UDP (*User Datagram Protocol*)**.

TCP

This kind of protocol **guarantees that sent data will arrive intact**, without information loss or data corruption, unless the connection fails. In this case, data packages are re-transmitted until they successfully arrive. A sequence of packages transmits a message. Each TCP package has an associated sequence of numbers that identify each package. These numbers allow the receiver system to reassemble the packages in the correct order regardless of the sequence they arrive. Also, using the same sequence of numbers the system can detect missing packages and require their retransmission. Figure 12.2 shows the structure of the TCP header for a better understanding of these correction mechanisms. Details about the meaning of all the fields in the TCP datagram can be found

in `https://tools.ietf.org/html/rfc3168#section-6.1`. We do not explain them here because we believe that it is out of the scope of this book. Some examples of TCP uses are: sending files via `FTP`, sending emails via `SMTP`, `POP3` or `HTTP`.

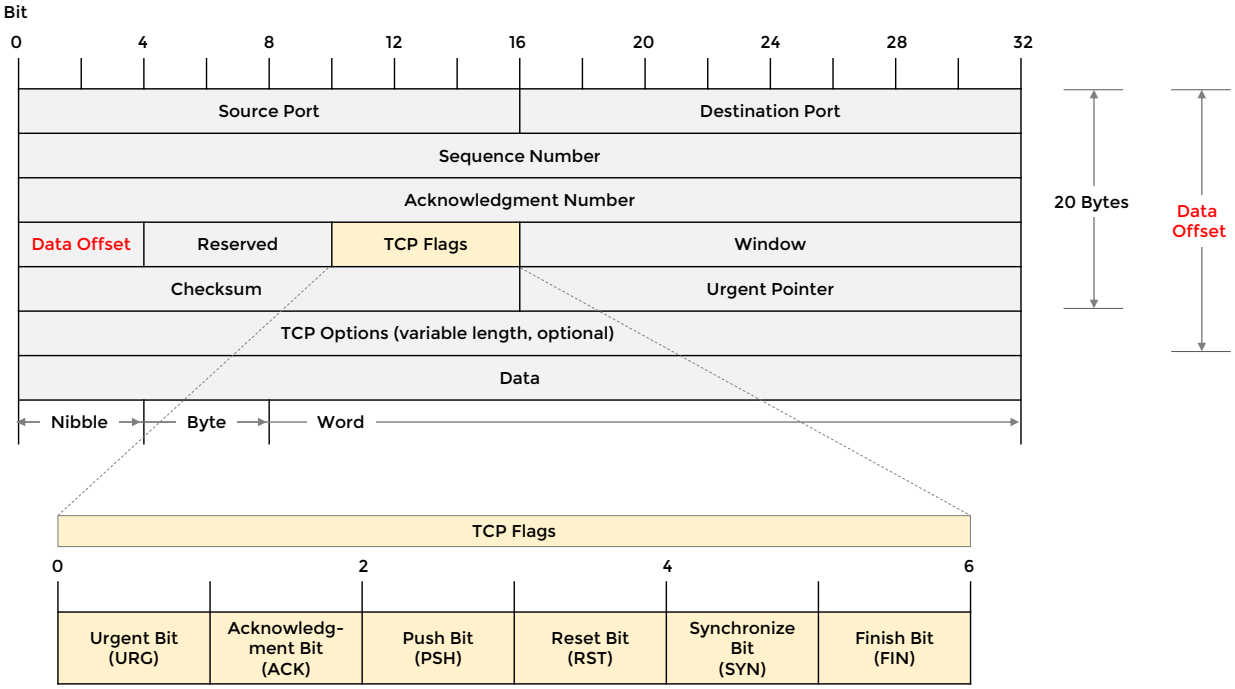


Figure 12.1: The figure shows the structure of the header in a TCP datagram. Note that it includes sections, such as the checksum, used to detect and fix errors during the transmission. The TCP flags can help us troubleshoot a connection.

UDP

UDP allows data transmission without establishing a connection. UDP packages have a header with enough information to be correctly identified and addressed through the network. Some examples are audio/video streaming, and online video games.

12.3 Sockets

To allow the communication among machines through the network, we need to generate an object which would be in charge of managing all the necessary information (hostname, address, port, etc.). *Sockets* are the Python objects that can handle the connection at a code level. To use sockets, we first need to import the `socket` module. To create a socket, we have to pass two arguments: the address family and socket type. There are two kinds of address families: `AF_INET`, for IPv4 addresses; and `AF_INET6`, for IPv6 addresses. Also there are two types of connections: `SOCK_STREAM`, for TCP connections; and `SOCK_DGRAM`, for UDP connections:

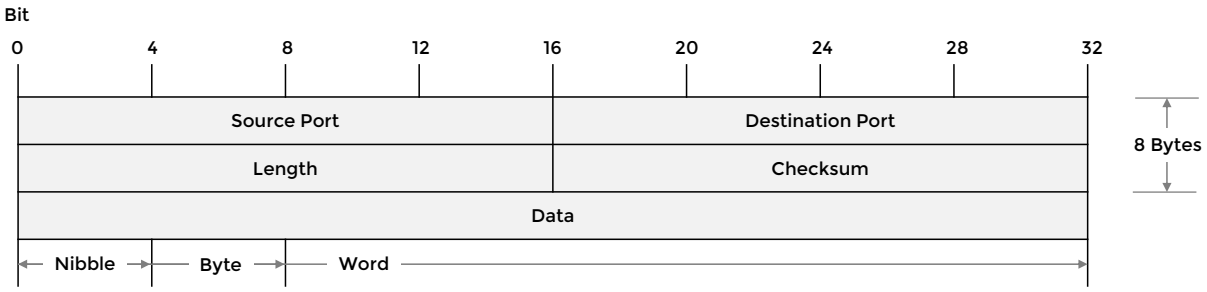


Figure 12.2: The figure shows the structure of the header in a UDP datagram. Note that it contains less information than TCP header. *Length* includes the number of bits used for header and data.

```
# create_socket.py

import socket

# Create TCP IPv4 socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print(s)
```

```
<socket.socket fd=3, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 0)>
```

12.4 Client-Server Architecture

Client-Server architecture corresponds to a network model between machines, in which some computers offer a service (servers), and others consume the service (clients). Figure 12.3 shows a diagram of this architecture. Clients must connect to a given server and use the necessary protocols to receive the requested service from it. A server must be constantly alert to potential client connections, to be able to deliver requested services when it receives connection attempts. Both sides in the client-server architecture accept TCP and UDP connections. However, both parties must use the same connection protocol to be able to communicate.

TCP client in Python

The following code shows an example of a TCP client in Python:

```
# tcp_client.py
```

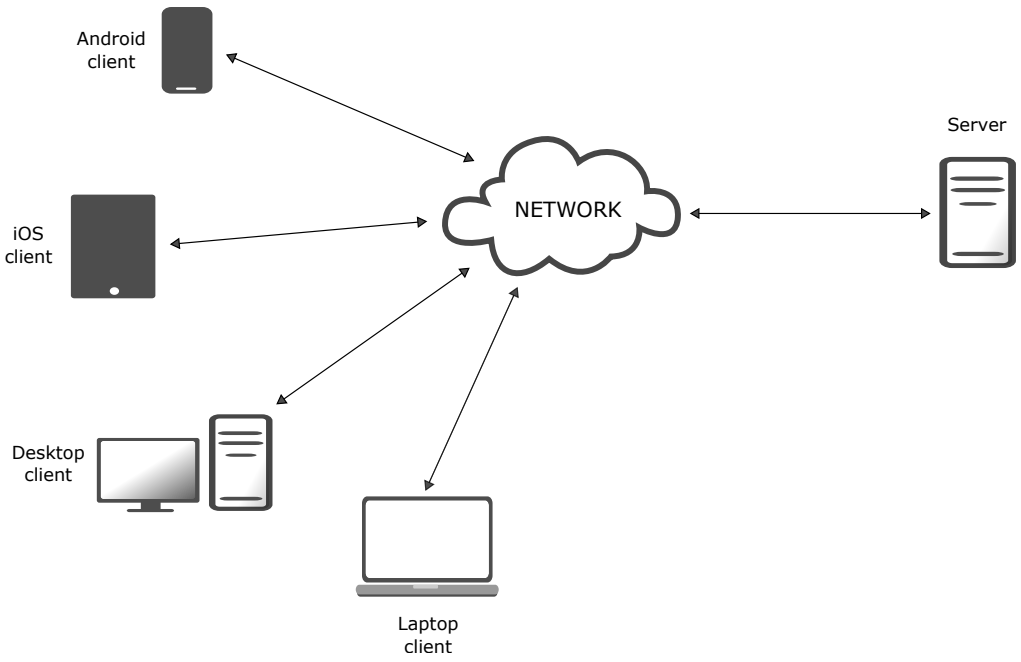


Figure 12.3: This figure shows the most common structure of connection between clients and a server. In general, the connection passes through several machines within the network before reaching the server.

```
2
3  import socket
4  import sys
5
6  MAX_SIZE = 1024
7
8  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9
10 try:
11     # Connect to a specify address
12     s.connect(("www.python.org", 80))
13
14     # Send a encoded string asking for the content of the page.
15     # Check https://www.w3.org/Protocols/rfc2616/rfc2616-sec14
16     #.html#sec14.23
17     # for possible protocol updates.
18     s.sendall("GET / HTTP/1.1\r\nHost: www.python.org\r\n\r\n"
```

```
                .encode('ascii'))

    # Receive the response. The argument indicates the buffer
    #size
    data = s.recv(MAX_SIZE)

    # Print received data after decoding
    print(data.decode('ascii'))

except socket.error:
    print("Connection error", socket.error)
    sys.exit()

finally:
    # Close connection
    s.close()
```

```
HTTP/1.1 301 Moved Permanently
Server: Varnish
Retry-After: 0
Location: https://www.python.org/
Content-Length: 0
Accept-Ranges: bytes
Date: Fri, 27 Jan 2017 21:08:53 GMT
Via: 1.1 varnish
Connection: close
X-Served-By: cache-dfw1838-DFW
X-Cache: HIT
X-Cache-Hits: 0
X-Timer: S1485551333.542024,VS0,VE0
Public-Key-Pins: max-age=600; includeSubDomains; pin-sha256=
"WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-sha256=
"5C8kvU039KouVrl52D0eZSGf4Onjo4Khs8tmyTlV3nU="; pin-sha256=
"5C8kvU039KouVrl52D0eZSGf4Onjo4Khs8tmyTlV3nU="; pin-sha256=
"lCppFqbkrlJ3EcVFAkeip0+44VaoJUymbnOaEUk7tEU="; pin-sha256=
"TUDnr0MEoJ3of7+YliBMBVFB4/gJsv5zO7IxD9+YoWI="; pin-sha256=
```

```
"x4QzPSC810K5/cMjb05Qm4k3Bw5zBn4lTdO/nEW/Td4=";
Strict-Transport-Security: max-age=63072000;
includeSubDomains
```

TCP server in Python

The following code shows an example of a TCP server in Python:

```
# tcp_server.py

import socket

# Create a TCP IPv4 socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 10001

# Bind the socker to the host and port
s.bind((host, port))

# We ask the operating system to start listening connections through
# the socket.
# The argument correspond to the maximun allowed connections.
s.listen(5)

count = 0
while True:
    # Stablish connection
    s_client, address = s.accept()
    print("Connection from:", address)

    # Prepare message
    message = "{}. Hi new friend!\n".format(count)

    # Change encoding and send
    s_client.send(message.encode("ascii"))
```

```
29
30      # Clonse current connection
31      s_client.close()
32      count += 1
```

Consider that the server and client must be executed in separated processes. Note that clients are not required to bind the host and the port, because the operating system implicitly does it in the `connect` method, assigning a random port for the client. The only case that needs a bind between a host and a particular port is when the server requires that the addresses of clients belong to a specific port range. In the server's case, the port must be linked to the address because clients must know how to find the server's exact position to connect to it. The `listen` method does not work if the address and the port are not linked. The following code is an example of a client that connects to the previously created server:

```
1  # tcp_server-listener.py
2
3  import socket
4
5  MAX_SIZE = 1024
6
7  s_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8
9  # Get local machine name
10 host = socket.gethostname()
11 port = 10001
12
13 s_client.connect((host, port))
14 data = s_client.recv(MAX_SIZE)
15 print(data.decode('ascii'))
16 s_client.close()
```

```
0. Hi new friend!
```

UDP client in Python

Given that the UDP protocol does not establish a connection, UDP communication code is much simpler. For example, to send a client message to a server, we only have to specify the server's address. Consider that the second argument

when creating the socket must be `SOCK_DGRAM`, for example:

```
# udp_client.py

import socket

MAXSIZE = 2048

# Create connection
server_name = socket.gethostbyname('localhost')
server_port = 25000
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Create message
message = "Hi, I'm sending this message."
target = (server_name, server_port)

# Send message
s.sendto(message.encode('ascii'), target)

# Optionally, we can get back sent information
# Also we can get the sender address
data, address = s.recvfrom(MAXSIZE)
print(data.decode('utf-8'))
```

```
Response for 127.0.0.1
```

Note that the string is encoded before being sent, to send bytes through the network. In Python 2 this was not necessary because the encoding was carried out automatically, but in Python 3 the encoding has to be performed explicitly. We can also receive the whole message fragmented in chunks. The following code shows how to assemble the message:

```
# fragmented.py

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
MAX_SIZE = 1024
```

```
7
8  fragments = []
9  finish = False
10 while not finish:
11     chunk = s.recv(MAX_SIZE)
12     if not chunk:
13         break
14     fragments.append(chunk)
15
16 # Joining original message
17 message = "".join(fragments)
```

UDP server in Python

To implement a server that sends messages using UDP, we only have to care about responding to the same address that sent the message. The following code shows an example of a server that communicates with the client implemented before:

```
1  # udp_server.py
2
3  import socket
4
5  MAXSIZE = 2048
6
7  server_name = socket.gethostbyname('localhost')
8  server_port = 25000
9
10 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
11 s.bind(("", server_port))
12
13 while True:
14     data, addr = s.recvfrom(MAXSIZE)
15     print(data.decode('ascii'))
16     response = "Response for {}".format(addr[0])
17     s.sendto(response.encode('utf-8'), addr)
```

```
Hi, I'm sending this message.
```

12.5 Sending JSON data

In the next example, we see how to generate a server that receives data and sends it back to the client. We then make a client that sends JSON data and prints it out after the server sends it back. Try this with two computers, one running as a server and the other as a client.

```
# json_server.py

import socket

MAX_SIZE = 1024

host = socket.gethostname()
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print(socket.gethostname())

s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
print('Connected:', addr)

while True:
    data = conn.recv(MAX_SIZE)
    if not data:
        break
    conn.sendall(data)

conn.close()
```

```
# json_client.py

import socket
import sys
import json

MAX_SIZE = 1024
```

```
8
9  server_host = socket.gethostname()
10 port = 12345
11 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 # Info to send as dictionary
14 info = {"name": "Johanna", "female": True}
15
16 # Create a string
17 message = json.dumps(info)
18
19 try:
20     s.connect((server_host, port))
21
22 except socket.gaierror as err:
23     print("Error: Connection error {}".format(err))
24     sys.exit()
25
26 # Send a message as bytes
27 s.sendall(bytes(message, "UTF-8"))
28
29 # Wait for response, then we decode it and convert it to JSON
30 data = json.loads(s.recv(MAX_SIZE).decode('UTF-8'))
31 print(data)
32 s.close()
```

12.6 Sending data with `pickle`

We can send any Python object serialized with `pickle`. The following code shows an example of how to connect to the previous server and to send `pickle` serialized data. When the bytes come back from the server, we de-serialize them and create a copy of the instance:

```
1 # pickle.py
2
3 import socket
4 import sys
```

```
5  import pickle_
6
7  MAX_SIZE = 1024
8
9  server_host = socket.gethostname()
10 port = 12345
11 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13
14 class Person:
15
16     def __init__(self, name, male):
17         self.name = name
18         self.male = male
19
20
21 person = Person("Sarah", True)
22 message = pickle_.dumps(person)
23
24 try:
25     s.connect((server_host, port))
26
27 except socket.gaierror as err:
28     print("Error: Connection error {}".format(err))
29     sys.exit()
30
31 s.sendall(message)
32 data = pickle_.loads(s.recv(MAX_SIZE))
33 print(data.name)
34 s.close()
```

Recall that pickle has a weakness in the sense that when we de-serialize a pickled object, we may execute arbitrary code on our computer. We can modify the server's code to make it do anything we want with the received data or to send anything back to the clients. We recommend you to connect two computers and play sending back and forth data to familiarize with sockets.

12.7 Hands-On Activities

Activity 12.1

Description

An internet startup needs to implement a bidirectional chat to communicate their employees. The application must give the option to become server or client, and you can assume that only a single server and client instance will connect. The communication flow must be synchronous, in other words, one user must wait for the other's response to reply. This application requires that sent messages appear on both endpoints, identifying the sender's username.

Chapter 13

Web Services

In the previous chapter, we have learned how to use sockets and protocols to establish communication through the client-server architecture. We also could transfer data between different computers. In this chapter, we will learn how to create a communication and data transference between computers by using the world wide web.

A *Web Service* is a grouping of client-server applications that communicate through the web using a specially designed protocol. We can see this type of service as a function or a black box that can be accessed by other programs via The Web. For example, let's consider the HTTP protocol used by internet browsers to get information from a website. Each time we execute any action within a web browser, a web server send a request to the browser. The server replies sending the required information (a web page for example), then our browser interprets that page and displays it to us in a friendly format. Web Services work in a similarly way, but the main difference is that the communication occurs between applications. Client and server must know the format of the exchanged information. It also helps to develop applications according to the hardware and keep the same communication structure.

Figure 13.1 shows the example of a house controlled remotely. This house is controllable by different computers with an internet connection through a web server. This server allows interaction between the house and other devices. One of the advantages of this model is the simplicity of the interaction between the applications, because of the independence of the languages used to implement the client and server. Each node can request the server to send or modify certain parameters of the house. Nodes may publish the information using JSON or XML or using any other format as well. A protocol bound used as an interface between two or more programs is known as **Application Programming Interface**, or **API** for short. APIs consist of a set of instructions that allows the applications to access the web.

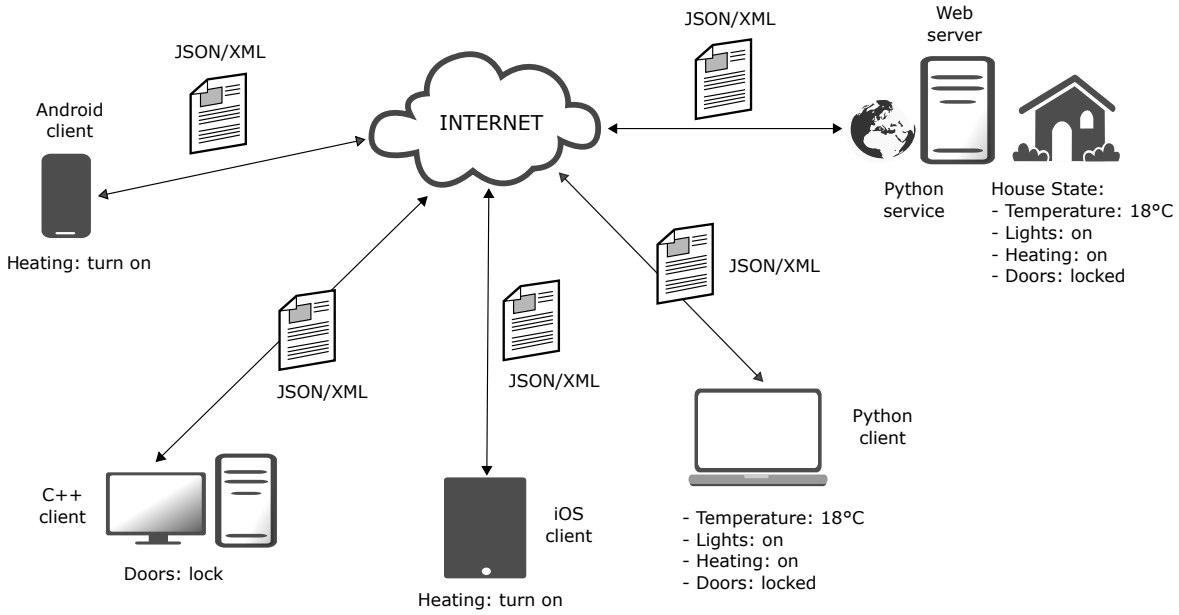


Figure 13.1: Diagram of a Web Service. Users interact with the house through clients software. In this example, clients request information and the Web Server provide a response in a JSON/XML format.

13.1 HTTP

A big part of the architecture in web services relies on the use of the HTTP or *Hypertext Transfer Protocol*. It is in charge of providing a layer to do transactions and allow communication between clients and servers. This protocol allows a higher level of communication when compared to TCP and UDP (see chapter 12). HTTP works as a *request-response* protocol in which the client performs a request and the server replies with the required information. It is a *state-less* protocol, which means that all commands are executed independently from the previous and future requests. The operation of this protocol depends on the definition of methods that indicate the action to perform on a particular resource. Resources could be existing data on the server such as files or entries in a database, or a dynamically generated output, among others. Version `HTTP/1.1` defines five methods, described in Table 13.1.

HTTP also consists of a set of **status codes** whereby they deliver information to the client about the result of its request. Table 13.2 shows an example of the HTTP message. We defer the read to the following link `http://www.w3schools.com/tags/ref\_httpmessages.asp` for more details about these codes.

Table 13.1: HTTP actions

HTTP method	Action
GET	Recovers a representation (information and meta-information) of a resource without changing anything in the server.
HEAD	Recovers only the meta-information (header) of a resource.
POST	Creates a resource.
PUT	Completely replaces a resource.
PATCH	Replaces selected attributes of a resource.
DELETE	Deletes a resource.

Table 13.2: Some common HTTP status codes

Status code	Description
200	OK. Successful request.
403	Forbidden. The request is accepted, but the server rejects it.
404	Not found. The requested resource was not found.
500	Internal server error.

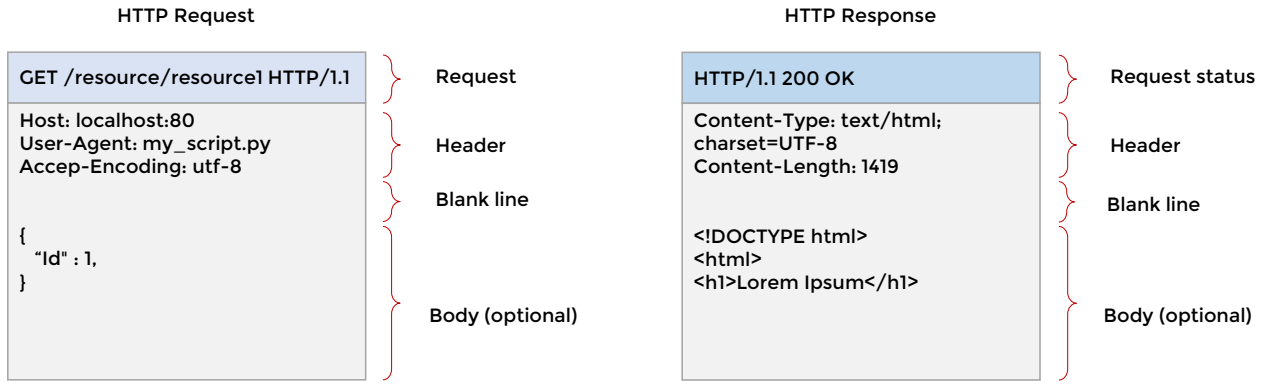


Figure 13.2: A simplified structure of the HTTP message. The blank line is intentionally added by the protocol.

13.2 REST architecture

One of the most used architectures for web service interaction is known as *Representational State Transfer* or simply **REST**. This structure uses standard HTTP methods to perform operations on the server. The calls to the server using REST reply in the format shown in the Figure 13.3, in which:

- The HTTP method corresponds to the action defined in the previous table.
- URI (Uniform Resource Identifier) is the identifier of a resource in a server.
- HTTP Version indicates the version of the protocol used by a request (`HTTP v1.1`).

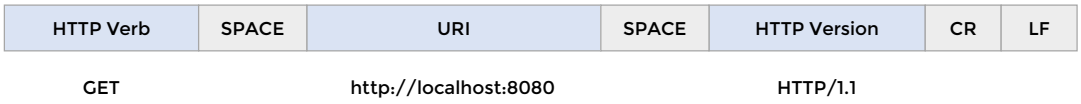


Figure 13.3: Representation of a request. Blank spaces exists between each part of the request. In the version `HTTP/1.1` the request requires the `CR` and `LF` characters at the end.

REST is straightforward and lightweight. The client and server implemented with REST and HTTP can take advantage of the entire internet infrastructure. In the Figure 13.4, we can see a diagram of the REST architecture.

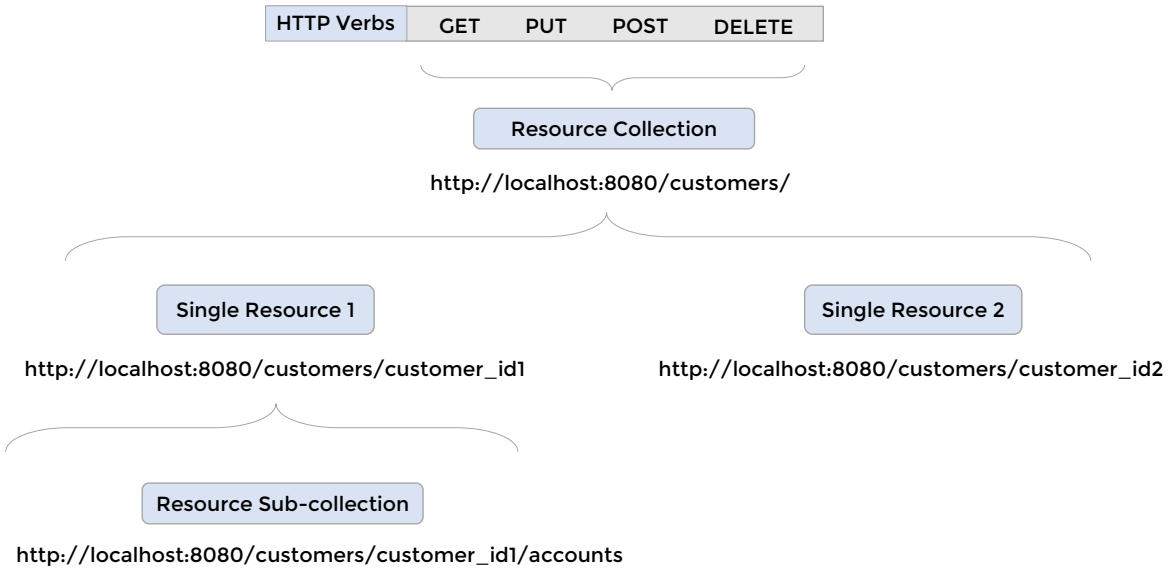


Figure 13.4: REST architecture schema. Links represent the resources. The hierarchy can be observed in the figure. There is no saved information about clients on the server. The request must contain all the necessary information to get the required resource.

13.3 Client-side Script

In this section, we see from the client viewpoint about how to perform requests to a server that hosts a web service. In Python, the `requests` library allows us to interact with several available web services. The library has the necessary HTTP methods for the REST structure. It also integrates JSON serializing methods as well.

To generate a request by means of GET, we use the `get(url)` method. For example, in the following script we generate a client that connects to the Google Image's API and recovers a query:

```
# request_google_images.py

import requests

# This URL contains the web service address
# and the required parameters

# %20 represents a 'space'

url = ('https://ajax.googleapis.com' +
       '/ajax/services/search/images?' +
       'v=1.0&q=copa%20america%20chile&as_filetype=jpg')

response = requests.get(url)
print(response.json())
```

```
{'responseData': None, 'responseDetails': 'This API is no longer available.',
'responseStatus': 403}
```

13.4 Server-side Script

To generate a web service we need to work on a *Web Framework* which lets us build dynamic websites and thus, manage the events by which the applications will interact. One of the most popular and significant framework choices for Python is *Django* (`https://www.djangoproject.com/`). However, for small applications, a micro-framework is more than enough. Here is a list of many smaller Python frameworks for web programming:

- Flask: `http://flask.pocoo.org/`
- Tornado: `http://www.tornadoweb.org/en/stable/`
- WebPy: `http://webpy.org/`
- CherryPy: `http://www.cherrypy.org/`
- Bottle: `http://bottlepy.org/docs/dev/index.html`

To exemplify the creation of a web service without loss of generality, we choose the **Flask** framework. This micro-framework is lightweight, easy to use and install, completely written in Python. Its form of encoding let us implement quickly REST-type web services.

The following example runs a server in the port `8080`. By default, the server runs locally in the IP address `127.0.0.1`, also referred as *localhost*:

```
# flask_server.py

import flask

app = flask.Flask(__name__)

# Originally, webservers used to have index.html as their main page.
#
# For example, http://mysite.com/index.html.
# By default, when no .html resource is requested
# on the URL root, it is assumed that will return
# the index.html.
#
# Nowadays this structure this folder structure
# is preserved but it is created dynamically catching the
# route and rendering anything we want.


# This route '/' determines the website root.
# Similar to the index.html document.
@app.route('/')
def index():
    return '<h1>Welcome to ur Web Service!</h1>' \
           '<p>HTML content</p>'


# Add a resource section
@app.route('/resources')
def resources_get():
```

```
30      return "<h1>Resources index</h1>"
31
32
33  # Resource section with arguments
34  @app.route('/resources/<resource_id>')
35  def resource_id_get(resource_id):
36      return '<p>Looking for resource with id: {}</p>'.format(resource_id)
37
38
39  if __name__ == '__main__':
40      # Start service as port 8080
41      app.run(port=8080)
```

Thanks to *Flask* we can start a dynamic web server using just a few lines of code. Once we run our script, we can see the result of our example in the browser, by typing in the address bar:

- `http://localhost:8080/` for the *root*.
- `http://localhost:8080/resources` to access our resources.
- `http://localhost:8080/resources/1` to access a resource created specifically with `id: 1`.

We can also execute our web service client with the following code:

```
1   # flask_client.py
2
3   import requests
4
5   r = requests.get('http://localhost:8080')
6   print('/: {}'.format(r.text))
7
8   r = requests.get('http://localhost:8080/resources')
9   print('/resources: {}'.format(r.text))
10
11  r = requests.get('http://localhost:8080/resources/1')
12  print('/resources/id: {}'.format(r.text))
```

```
/: <h1>Welcome to ur Web Service!</h1><p>HTML content</p>
/resources: <h1>Resources index</h1>
/resources/id: <p>Looking for resouce with id: 1</p>
```

13.5 Request

From the server's side, requests are calls from the client in which it is possible to receive arguments for the corresponding resource. These arguments are sent from the client through the methods: `GET`; `POST`; `PUT` and `DELETE`. *Flask* manages the sent data from the calls via the `request` class. Let's assume that we have the following service that allows sending two values to the service and define an operation using the type of resource called:

```
1  # request.py
2
3  import flask
4
5  app = flask.Flask(__name__)
6
7
8  def pow(v1, v2):
9      return v1 ** v2
10
11
12 def add(v1, v2):
13     return v1 + v2
14
15 # Server functions
16 fun_handle = {'pow': pow, 'add': add}
17
18
19 @app.route('/api/<api_id>')
20 def api_get(api_id):
21     # Request.args contains a dictionary with the
22     # arguments that were sent by the client
23     args = flask.request.args
24     if 'v1' not in args and 'v2' not in args:
25         return 'Error: No values were found'
```

```
26      else:
27          # Parse string to integer
28          v1 = int(args['v1'])
29          v2 = int(args['v2'])
30
31          return '{0}: {1}'.format(api_id, fun_handle[api_id](v1, v2))
32
33
34  if __name__ == '__main__':
35      # Start service as port 8080
36      app.run(port=8080)
```

Using our client, we can send arguments to the different functions of the service that the server is running. To transfer the parameters, we use the `params` keyword inside the `get` method.

```
1  # request_usage.py
2
3  import requests
4
5  r = requests.get('http://localhost:8080/api/pow',
6                   params={'v1': '10', 'v2': '2'})
7
8  print(r.text)
9
10 r = requests.get('http://localhost:8080/api/add',
11                  params={'v1': '10', 'v2': '2'})
12
13 print(r.text)
```

```
pow: 100
add: 12
```

13.6 Request Data

Aside from sending values as arguments in the request, it is also possible to send data to the server in JSON format, or plain text, by using the POST method. Using POST allows, aside from many things, to not leave any local information

(cache) from the sent data. It also does not let the data get exposed alongside the URI. Visually, this would be like seeing the data alongside the address in the search bar of our browser. The negotiation of the sent contents is done through the `HEADER`:

```
# request_post_server.py

import flask
import json

app = flask.Flask(__name__)


# This time only POST methods
@app.route('/upload', methods=['POST'])
def api_post():
    req = flask.request
    if req.headers['Content-Type'] == 'application/json':
        with open('post.txt', 'w') as fid:
            json.dump(req.json, fid)

        # Send echo to the client
        return "Echo: {}".format(json.dumps(req.json))


if __name__ == '__main__':
    # Start service as port 8080
    app.run(port=8080)
```

```
# request_post_client.py

import requests
import json

# Data as a form
form = {'id': 1,
        'name': 'Guido',
        'last_name': 'Van Rossum'}
```

```
10
11 # Header declares content type
12 header = {'Content-Type': 'application/json'}
13
14 # Make the request
15 r = requests.post('http://localhost:8080/upload',
16                    headers=header,
17                    data=json.dumps(form))
18
19 # Status code, 200 is 'OK'
20 print(r.status_code)
21 print(r.text)
```

```
200
Echo: {"last_name": "Van Rossum", "id": 1, "name": "Guido"}
```

13.7 Response

As we saw at the beginning of this chapter, to communicate applications, it is easier to use a format or protocol to send information back and force. Within the most used formats, JSON and XML are among the most popular. In the specific case of *Flask*, server response is managed via the `Response` class. Now, we see a serialization example of the answer of a server through JSON:

```
1  # request_response_server.py
2
3  import flask
4  import json
5
6
7  def get_id():
8      pid = 0
9      while True:
10         yield pid
11         pid += 1
12
13 pid = get_id()
```

```
14
15
16 class Person:
17     def __init__(self, name, number):
18         self.name = name
19         self.number = number
20         self.id = next(pid)
21
22
23 app = flask.Flask(__name__)
24
25
26 # Response to GET method at /api route
27 @app.route('/api', methods=['GET'])
28 def api_echo():
29     person = Person('Jason Kruger', '20.000.000-0')
30     return flask.Response(json.dumps(person.__dict__), status=200)
31
32
33 if __name__ == '__main__':
34     # Start service as port 8080
35     app.run(port=8080)
```

```
1 # request_response_client.py
2
3 import requests
4
5 r = requests.get('http://localhost:8080/api')
6 print('{}'.format(r.json()))
7
8 r = requests.get('http://localhost:8080/api')
9 print('{}'.format(r.json()))
```

```
{'name': 'Jason Kruger', 'number': '20.000.000-0', 'id': 2}
{'name': 'Jason Kruger', 'number': '20.000.000-0', 'id': 3}
```

13.8 Other architectures for Web Services

Other architectures to implement web services are XML/RCP and SOAP. Both use HTTP and XML to transfer data.

The **XML-RPC** protocol (*XML for Remote Procedure Call*) use HTTP to send requests to the server. Just like in REST, the client is an application that makes a call to a server giving arguments, which then returns a value. The use of XML allows the serialization of structures or objects as input and output of the resources. Unlike REST, which uses representations of resources, XML-RCP performs calls to methods on the server.

The **SOAP** protocol (*Simple Object Access Protocol*) uses HTTP or SMTP (Simple Mail Transfer Protocol) as transport and only XML to define the messages. SOAP allows the use of communication between different platforms, achieving the specification for *Web Services Description Language (WSDL)* and *Universal Description, Discovery, and Integration (UDDI)*. Just like XML-RCP, in the message, it defines the methods that must be executed on the server. Figure 13.5 shows the interaction between the client and server, following the SOAP architecture.

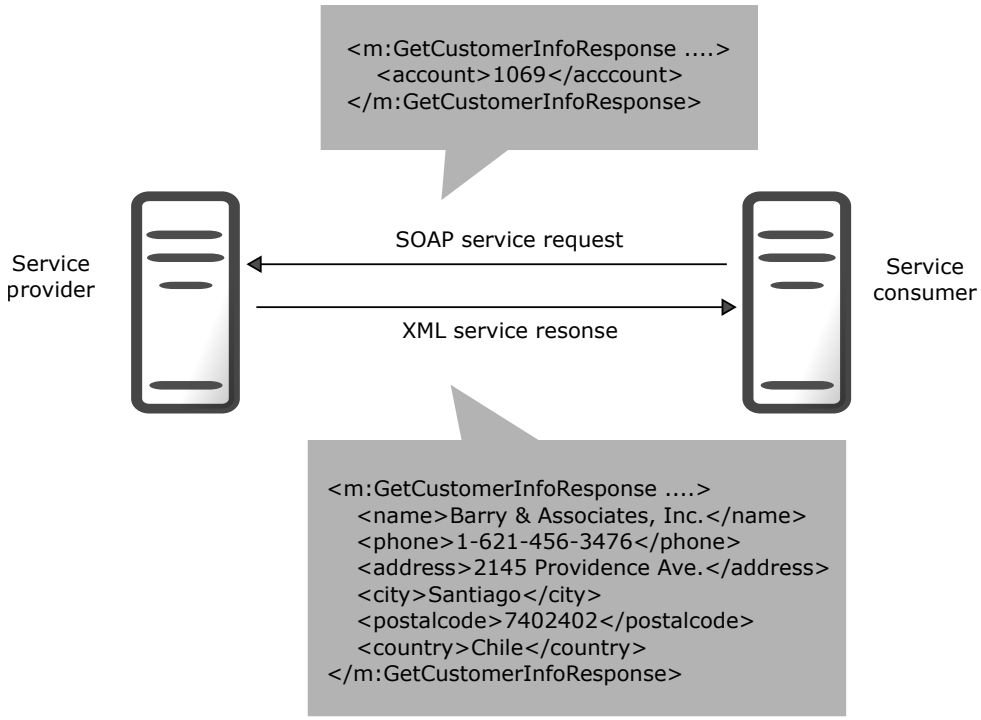


Figure 13.5: SOAP interaction

The following is an example of a SOAP message (Source: `https://en.wikipedia.org/wiki/SOAP`):

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
```

```
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Chapter 14

Graphical User Interfaces

So far, we have developed programs that interact with the user through the command line, where the user has to call a Python program by typing its name and adding the sequence of arguments.

Having a visual environment for the program variables and results extremely simplify the interaction between the user and the application. This kind of environments are known as a *Graphical User Interfaces (GUI)*. Graphical interfaces are present in various types of devices and platforms, such as web form or a smartphone application. Most, if not all, graphical user interface based applications use an *event management based architecture*. Applications operated by command line perform data input and output at specific times and circumstances established within the program. However, every time we interact with an application graphically, the program does not know beforehand when actions will occur. The user may enter data, press a key, move the mouse, or click on any widget within the interface. Therefore, the code must be written to respond to all these events. It is always possible to design a graphical interface from the beginning to the end by interacting directly with each pixel. Never the less, there are now optimized modules that provide generic graphical elements such as buttons, bars, text boxes, and calendars. These modules greatly facilitate the development of graphical interfaces. In this course, we will focus on the use of *PyQt*.

14.1 PyQt

PyQt is a Python library that includes several modules that help with the development of graphical interfaces, here we describe some of the modules included in *PyQt*:

- QtCore: includes non-GUI functionalities like file and directory management, time, and URLs, among others.
- QtGui: includes visual components such as buttons, windows, drop-down lists, etc.

- QtNetwork: provides classes to program graphical applications based on TCP/IP, UDP, and some other network protocols.
- QtXml: includes XML file handling functionalities.
- QtSvg: provides classes to display vector graphics files (SVG).
- QtOpenGL: contains functions for 3D and 2D graphics rendering using OpenGL.
- QtSql: includes functionalities for working with SQL databases.

To create a window we use the *QtGui* module. The first step is to build the application that will contain the window and all its elements or *Widgets*. This procedure is done through the `QApplication` class, which includes event loop, application initialization and closing, input arguments handling, among other responsibilities. For every application that *PyQt* uses, there is only one `QApplication` object regardless of the number of windows that this application has.

```
1  # codes_1.py
2
3  from PyQt4 import QtGui
4
5
6  class MiForm(QtGui.QWidget):
7      # The next line defines the window geometry.
8      # Parameters: (x_top_left, y_top_left, width, height)
9      self.setGeometry(200, 100, 300, 300)
10     self.setWindowTitle('My First Window')  # Optional
11
12
13 if __name__ == '__main__':
14     app = QtGui.QApplication([])
15     form = MiForm()
16     form.show()
17     app.exec_()
```

The `QtGui.QWidget` class from which the `MyForm` class descends, is the base class for all objects in *PyQt*. The constructor for this class has no default parents, in which case corresponds to an empty window. We also have to define the window properties, so far only defined in memory. To display the window on the screen, we must use the

`show()` method. Finally, the `exec_()` method executes the main loop to perform the event detection. The result of the above code corresponds to the clear interface as shown in Figure 14.1.

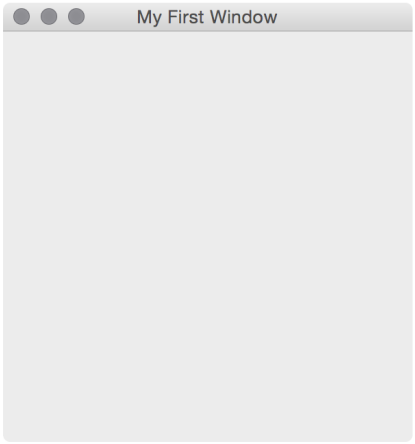


Figure 14.1: Example of an empty window generated by PyQt.

In *PyQt* there are useful objects or Widgets to control information input and output in graphical interfaces. These are labels and text boxes. Labels are used to display form variables or static strings. In *PyQt* these are represented by the `QLabel` class. Text boxes also allow text handling in the interface, primarily as a means of entering data into the form. In *PyQt* interface they are created by `QLineEdit` class:

```
1  # codes_2.py
2
3  from PyQt4 import QtGui
4
5
6  class MiForm(QtGui.QWidget):
7      def __init__(self):
8          super().__init__()
9          self.init_GUI()
10
11     def init_GUI(self):
12         # Once the form is called, this method initializes the
13         # interface and all of its elements and Widgets
14
15         self.label1 = QtGui.QLabel('Text:', self)
16         self.label1.move(10, 15)
17
```

```
18          self.label2 = QtGui.QLabel('This label is modifiable', self)
19          self.label2.move(10, 50)
20
21          self.edit1 = QtGui.QLineEdit('', self)
22          self.edit1.setGeometry(45, 15, 100, 20)
23
24          # Adds all elements to the form
25          self.setGeometry(200, 100, 200, 300)
26          self.setWindowTitle('Window with buttons')
27
28
29  if __name__ == '__main__':
30      app = QtGui.QApplication([])
31
32      # A window that inherits from QMainWindow is created
33      form = MiForm()
34      form.show()
35      app.exec_()
```

The code above clarifies how to use `Labels` and `LineEdits` within the GUI. Figure 14.2 shows the results.

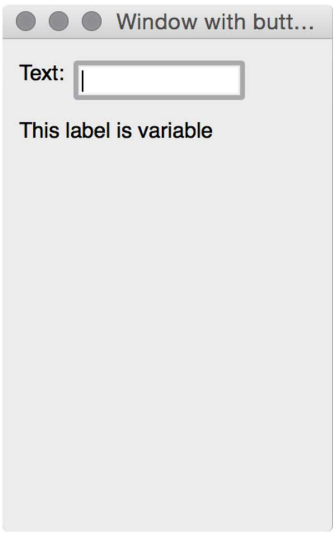


Figure 14.2: Example of a QLabel within a Widget.

PyQt also includes several useful graphical objects to control the interface. The most basic is the `PushButton(label, father, function)` element, which embeds a button in the window. The result

generated by the code below is a window with a button as shown in the Figure 14.3.

```
# codes_3.py

from PyQt4 import QtGui


class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        # Once the form is called, this method initializes the
        # interface and all of its elements and Widgets
        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.label2 = QtGui.QLabel('Write the answer here', self)
        self.label2.move(10, 50)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        # The use of the & symbol at the start of the text within
        # any button or menu makes it so the first letter is shown
        # in bold font. This visualization may depend on the used
        # platform.
        self.button1 = QtGui.QPushButton('&Process', self)
        self.button1.resize(self.button1.sizeHint())
        self.button1.move(5, 70)

        # Adds all elements to the form
        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Window with buttons')
        self.show()
```

```
35
36
37 if __name__ == '__main__':
38     app = QtGui.QApplication([])
39
40     # A window that inherits from QMainWindow is created
41     form = MyForm()
42     form.show()
43     app.exec_()
```

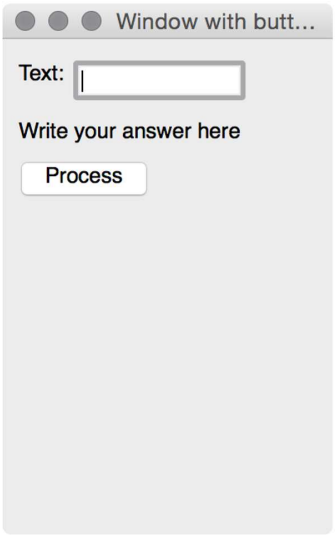


Figure 14.3: A simple window with a button.

Main Window

Windows created by `QWidget` correspond to windows that may contain other Widgets. *PyQt* offers a more complete type of window called `MainWindow`. It creates the standard skeleton of an application including a status bar, toolbar, and menu bar, as shown in the Figure 14.4.

The **status bar** allows us to display application status information. To create this, the `statusBar()` method (belongs to `QApplication` class) is used. Messages in the status bar are updated when the user interacts with the rest of the application. The **menu bar** is a typical part of a GUI-based application. It corresponds to a group of structured and logically grouped commands inside of menus. The **toolbar** provides quick access to the most frequently used commands. Finally, the **Central Widget** or core content area corresponds to the body of the window. It may contain any Widget in *QtGui*, as well as one of the forms created in the previous examples. To add this Widget or form to

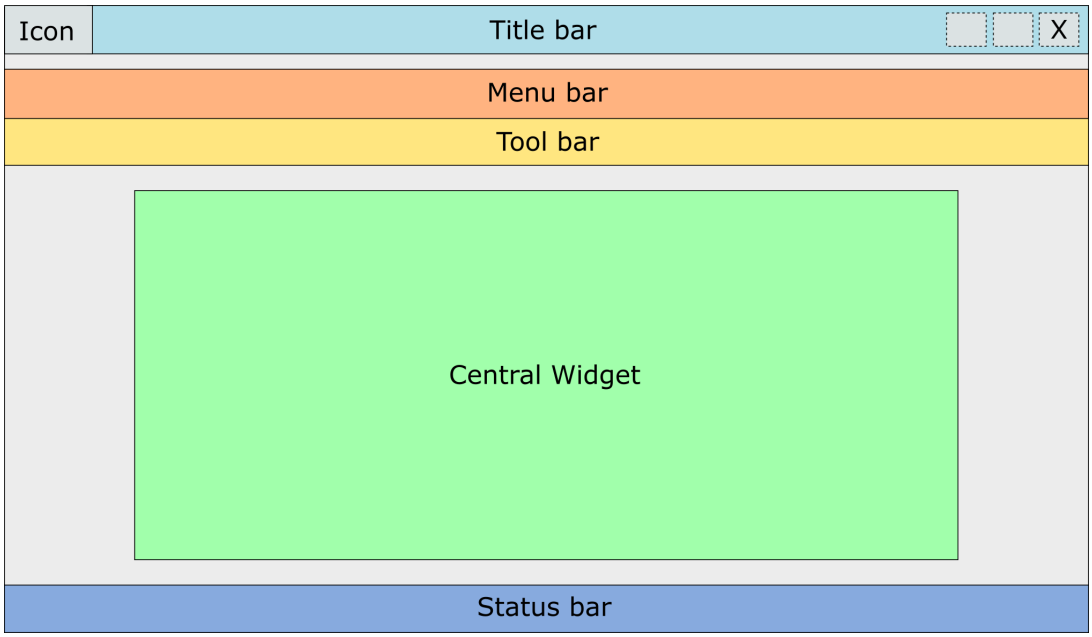


Figure 14.4: Diagram of the classic `MainWindow` layout.

the central Widget, the `setCentralWidget(Widget)` method must be used. The next example shows how to integrate the elements described in the main window:

```
1  # codes_4.py
2
3  from PyQt4 import QtGui
4
5
6  class MainForm(QtGui.QMainWindow):
7      def __init__(self):
8          super().__init__()
9
10         # Configures window geometry
11         self.setWindowTitle('Window with buttons')
12         self.setGeometry(200, 100, 300, 250)
13
14         # Action definitions
15         see_status = QtGui.QAction(QtGui.QIcon(None), '&Change '
16                                                       'Status', self)
17         see_status.setStatusTip('This is a test item')
```

```
18          see_status.triggered.connect(self.change_status_bar)
19
20          exit = QtGui.QAction(QtGui.QIcon(None), '&Exit', self)
21          # We can define a key combination to execute each command
22          exit.setShortcut('Ctrl+Q')
23          # This method shows the command description on the status bar
24          exit.setStatusTip('End the app')
25          # Connects the signal to the slot that will handle this event
26          exit.triggered.connect(QtGui.qApp.quit)
27
28          # Menu and menu bar creation
29          menubar = self.menuBar()
30          file_menu = menubar.addMenu('&File')  # first menu
31          file_menu.addAction(see_status)
32          file_menu.addAction(exit)
33
34          other_menu = menubar.addMenu('&Other Menu')  # second menu
35
36          # Includes the status bar
37          self.statusBar().showMessage('Ready')
38
39          # Sets the previously created form as the central widged
40          self.form = MyForm()
41          self.setCentralWidget(self.form)
42
43      def change_status_bar(self):
44          self.statusBar().showMessage('Status has been changed')
45
46
47  class MyForm(QtGui.QWidget):
48      def __init__(self):
49          super().__init__()
50          self.init_GUI()
51
52      def init_GUI(self):
```

```
53          self.label1 = QtGui.QLabel('Text:', self)
54          self.label1.move(10, 15)
55
56          self.label2 = QtGui.QLabel('Write the answer here', self)
57          self.label2.move(10, 50)
58
59          self.edit1 = QtGui.QLineEdit('', self)
60          self.edit1.setGeometry(45, 15, 100, 20)
61
62          self.button1 = QtGui.QPushButton('&Process', self)
63          self.button1.resize(self.button1.sizeHint())
64          self.button1.move(5, 70)
65
66          self.setGeometry(200, 100, 200, 300)
67          self.setWindowTitle('Window with buttons')
68          self.show()
69
70      def button_pressed(self):
71          sender = self.sender()
72          self.label3.setText('Signal origin: {0}'.format(sender.text()))
73          self.label3.resize(self.label3.sizeHint())
74
75      def button1_callback(self):
76          self.label2.setText('{}'.format(self.edit1.text()))
77          self.label2.resize(self.label2.sizeHint())
78
79
80  if __name__ == '__main__':
81      app = QtGui.QApplication([])
82
83      form = MainForm()
84      form.show()
85      app.exec_()
```

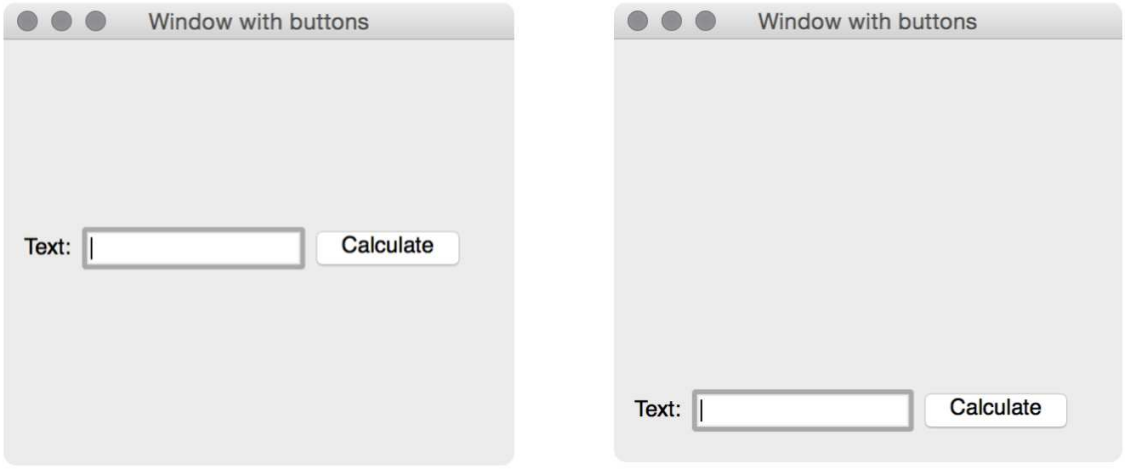
14.2 Layouts

Layouts allow a more flexible and responsive way to manage and implement Widget distribution in the interface window. Each Widget's `move(x, y)` method allows absolute positioning of all objects in the window. However, it has limitations, such as; Widget position does not change if we resize the window, and Application's look will vary on different platforms or display settings.

To avoid redoing the window for better distribution, we use **box layouts**. Two basic types allow Widget horizontal and vertical alignment: `QtGui.QHBoxLayout()` and `QtGui.QVBoxLayout()`. In both cases, Widgets are distributed within the layout occupying all available space, even if we resize the window. Objects must be added to each layout by the `addWidget` method. Finally, the box layout must be loaded to the window as `self.setLayout()`. We can add the vertical alignment of objects by including the horizontal layout within a vertical layout. The following code shows an example of how to create a layout such that three Widgets are aligned in the bottom right corner. Figure 14.5 shows the output:

```
1  # codes_5.py
2
3  from PyQt4 import QtGui
4
5
6  class MainForm(QtGui.QMainWindow):
7      def __init__(self):
8          super().__init__()
9
10         # Window geometry
11         self.setWindowTitle('Window with buttons')
12         self.setGeometry(200, 100, 300, 250)
13
14         self.form = MiForm()
15         self.setCentralWidget(self.form)
16
17
18 class MiForm(QtGui.QWidget):
19     def __init__(self):
20         super().__init__()
21         self.init_GUI()
```

```
22
23      def init_GUI(self):
24          self.label1 = QtGui.QLabel('Text:', self)
25          self.label1.move(10, 15)
26
27          self.edit1 = QtGui.QLineEdit('', self)
28          self.edit1.setGeometry(45, 15, 100, 20)
29
30          self.button1 = QtGui.QPushButton('&Calculate', self)
31          self.button1.resize(self.button1.sizeHint())
32
33          # QHBoxLayout() and QVBoxLayout() are created and added to the
34          # Widget list by using the addWidget() method. The stretch()
35          # method includes a spacing that expands the layout towards
36          #  the right and downwards.
37          hbox = QtGui.QHBoxLayout()
38          hbox.addStretch(1)
39          hbox.addWidget(self.label1)
40          hbox.addWidget(self.edit1)
41          hbox.addWidget(self.button1)
42
43          vbox = QtGui.QVBoxLayout()
44          vbox.addStretch(1)
45          vbox.addLayout(hbox)
46
47          # The vertical layout contains the horizontal layout
48          self.setLayout(vbox)
49
50
51  if __name__ == '__main__':
52      app = QtGui.QApplication([])
53
54      form = MainForm()
55      form.show()
56      app.exec_()
```



(a) (b)

Figure 14.5: This figure shows the possible results after executing the code above. (a) Shows only using `QHBoxLayout`. (b) Shows using `QHBoxLayout` and `QVBoxLayout`.

PyQt includes a class to distribute widgets in a matrix-like grid within the window, called `QGridLayout()`. This type of layout divides the window space into rows and columns. Each Widget must be added to a cell in the grid by using the `addWidget(Widget, i, j)` method. For example, the following code shows an example to create a matrix similar to a mobile phone keypad buttons. The output can be seen in Figure 14.6.

```
1  # codes_6.py
2
3  class MyForm(QtGui.QWidget):
4      def __init__(self):
5          super().__init__()
6          self.init_GUI()
7
8      def init_GUI(self):
9          # Creating the grid that will position Widgets in a matrix
10         # like manner
11         grid = QtGui.QGridLayout()
12         self.setLayout(grid)
13
14         values = ['1', '2', '3',
15                   '4', '5', '6',
16                   '7', '8', '9',
```

```
17                  '*', '0', '#']
18
19          positions = [(i, j) for i in range(4) for j in range(3)]
20
21          for _positions, value in zip(positions, values):
22              if value == '':
23                  continue
24
25              # The * symbol allows unpacking _positions as
26              # independent arguments
27              button = QtGui.QPushButton(value)
28              grid.addWidget(button, *_positions)
29
30          self.move(300, 150)
31          self.setWindowTitle('Calculator')
32          self.show()
```

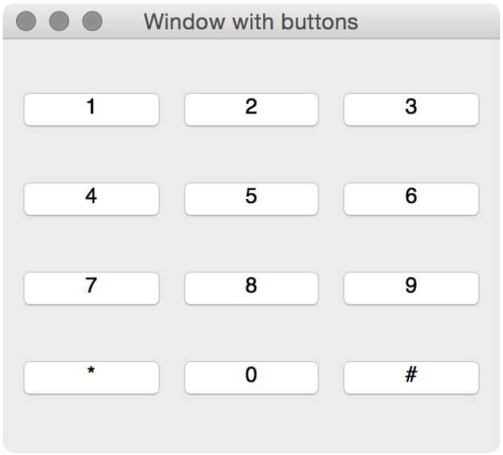


Figure 14.6: Example of numeric keypad using `QGridLayout` to organize the buttons in a grid.

14.3 Events and Signals

Graphical interfaces are applications focused mainly on handling events. This strategy allows detecting user actions on the interface asynchronously. The same application can generate events A. In *PyQt* these events are detected once the application enters the main loop started by the `exec_()` method. Figure 14.6 shows a flowchart comparison

between a program with a linear structure and a program using GUI-based event handling. In this model there are three fundamental elements:

- The source of the event: corresponds to the object that generates the change of state or that generates the event
- The event object: the object that encapsulates the change of status through the event.
- The target object: the object to be notified of the status change

Under this model, the event source delegates the task of managing the event to the target object. *PyQt*, on its 4th version, uses the *signal* and *slot* mechanism to handle events. Both elements are used for communication between objects. When an event occurs, the activated object generates signals, and any of those signals calls a slot. Slots can be any callable Python object.

Below, we see a modification to the previous program so that it generates a call to the `boton1_callback()` function, after `button1` is pressed. This is accomplished using the event to send the signal. In the case of buttons, the signal corresponds to the *clicked* method. By using the `connect()` method, communication between objects involved in the event is set. This method receives a Python callable function, i.e., `boton1_callback` without `()`.

```
1  # codes_7.py
2
3  class MyForm(QtGui.QWidget):
4      def __init__(self):
5          super().__init__()
6          self.init_GUI()
7
8      def init_GUI(self):
9          self.label1 = QtGui.QLabel('Text:', self)
10         self.label1.move(10, 15)
11
12         self.label2 = QtGui.QLabel('Write your answer here', self)
13         self.label2.move(10, 50)
14
15         self.edit1 = QtGui.QLineEdit('', self)
16         self.edit1.setGeometry(45, 15, 100, 20)
17
18         # Connecting button1 signal to other object
```

```
19          self.button1 = QtGui.QPushButton('&Process', self)
20          self.button1.resize(self.button1.sizeHint())
21          self.button1.move(5, 70)
22          # This object MUST be callable. self.button1_callback()
23          # would not work.
24          self.button1.clicked.connect(self.button1_callback)
25
26          self.button2 = QtGui.QPushButton('&Exit', self)
27          self.button2.clicked.connect(
28              QtCore.QCoreApplication.instance().quit)
29          self.button2.resize(self.button2.sizeHint())
30          self.button2.move(90, 70)
31
32          self.setGeometry(200, 100, 200, 300)
33          self.setWindowTitle('Window with buttons')
34
35      def button1_callback(self):
36          # This method handles the event
37          self.label2.setText(self.edit1.text())
```

14.4 Sender

Sometimes we need to know which of the objects on the form sent a signal. *PyQt* offers the `sender()` method. We can see an example by adding a new label to the form that will display the name of the widget that sends the signal:

```
1  # codes_8.py
2
3  def init_GUI(self):
4      self.label1 = QtGui.QLabel('Text:', self)
5      self.label1.move(10, 15)
6
7      self.label2 = QtGui.QLabel('Write your answer here:', self)
8      self.label2.move(10, 50)
9
10     self.label3 = QtGui.QLabel('Signal origin:', self)
11     self.label3.move(10, 250)
```

```
12
13      self.edit1 = QtGui.QLineEdit('', self)
14      self.edit1.setGeometry(45, 15, 100, 20)
15
16      self.button1 = QtGui.QPushButton('&Process', self)
17      self.button1.resize(self.button1.sizeHint())
18      self.button1.move(5, 70)
19      self.button1.clicked.connect(self.button1_callback)
20      self.button1.clicked.connect(self.button_pressed)
21
22      self.button2 = QtGui.QPushButton('&Exit', self)
23      self.button2.clicked.connect(QtCore.QCoreApplication.instance().quit)
24      self.button2.resize(self.button2.sizeHint())
25      self.button2.move(90, 70)
26
27      self.setGeometry(200, 100, 300, 300)
28      self.setWindowTitle('Window with buttons.')
29      self.show()
30
31
32  def button_pressed(self):
33      # This method registers the object sending the signal and shows it in
34      # label3 by using the sender() method
35      sender = self.sender()
36      self.label3.setText('Signal origin: {0}'.format(sender.text()))
37      self.label3.resize(self.label3.sizeHint())
38
39
40  def button1_callback(self):
41      self.label2.setText(self.edit1.text())
```

14.5 Creating Custom Signals

In *PyQt* it is also possible to define user-customized signals. In this case, we must create the object that will host the new signal. These signals are a subclass of `QtCore.QObject`. Within the object the new signal is created as an

instance of the object `QtCore.pyqtSignal()`. Then, the signal and its handling functions if required, must be created in the form. The example below shows a simple way to generate a new signal that activates when one of the buttons is pressed. To emit the signal the `emit()` method inherited from `pyqtSignal()` is used:

```
1  # codes_9.py
2
3  import sys
4  from PyQt4 import QtGui, QtCore
5
6
7  class MySignal(QtCore.QObject):
8      # This class defines the new signal 'signal_writer'
9      signal_writer = QtCore.pyqtSignal()
10
11
12 class MyForm(QtGui.QWidget):
13     def __init__(self):
14         super().__init__()
15         self.initialize_GUI()
16
17     def initialize_GUI(self):
18         self.s = MySignal()
19         self.s.signal_writer.connect(self.write_label)
20
21         self.label1 = QtGui.QLabel('Label', self)
22         self.label1.move(20, 10)
23         self.resize(self.label1.sizeHint())
24
25         self.setGeometry(300, 300, 290, 150)
26         self.setWindowTitle('Signal Emitter')
27         self.show()
28
29     def mousePressEvent(self, event):
30         # This method handles when any of the mouse buttons is pressed. It is
31         # defined by default within the app. It can be overwritten according
32         # to how the event should be handled in each app.
```

```
33          self.s.signal_writer.emit()
34
35      def write_label(self):
36          self.label1.setText('Mouse was clicked')
37          self.label1.resize(self.label1.sizeHint())
38
39
40  def main():
41      app = QtGui.QApplication(sys.argv)
42      ex = MyForm()
43      sys.exit(app.exec_())
44
45
46  if __name__ == '__main__':
47      main()
```

14.6 Mouse and Keyboard Events

Another way to generate events is through the keyboard and mouse. These events can be handled by overriding two methods defined in the `MainForm`: `mousePressEvent()` and `keyPressEvent()`.

```
1  # codes_10.py
2
3  def keyPressEvent(self, event):
4      self.statusBar().showMessage('Key pressed {}'.format(event.text()))
5
6
7  def mousePressEvent(self, *args, **kwargs):
8      self.statusBar().showMessage('Mouse click')
```

14.7 QT Designer

When GUIs have few Widgets, it is easy to create them manually by adding each one with code. However, when the interface includes a larger number of objects, interactions or controls, the code gets longer and hard to maintain. Fortunately, *PyQt* provides a tool called *QT Designer* that allows building the graphical interface visually. *Qt Designer*

allows to create each widget in the interface and also gives you control over all the properties of Widgets. Figure 14.7 shows the main view of *Qt Designer*.

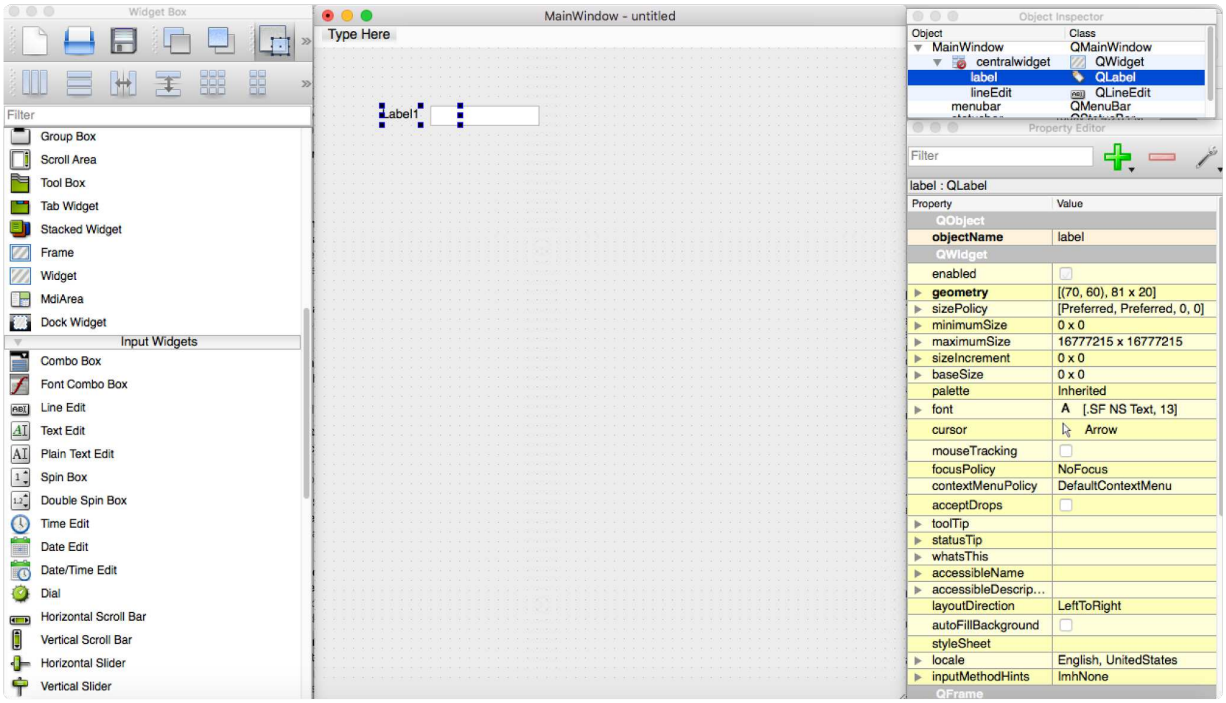


Figure 14.7: Example of *QtDesigner* application's main view.

All *QTDesigner*'s actions can be controlled from the application menu. There are four major sections in the working window. On the left side is the Widget Box containing all Widgets that can be added to the form, sorted by type. On the right side, we can find the Object Inspector which allows displaying the hierarchy of widgets that exist in the form. Following the Object Inspector is the property editor. Finally, in the center we may find the central Widget that can be a simple window or form, or a more complex Widget:

Once we add a Widget to the interface, a default name is assigned to the `objectName` field in the property editor. This name can be changed directly, as shown in figure 14.9. The same name will be the one used to reference the object from the Python code that handles the interface.

A natural way to see the result without embedding it in the final code is to use Qt Designer's preview mode, accessed by pressing **Ctrl + R**. In this mode the interface created is fully functional. Once the interface is complete, it must be saved. At that time a file with the `.ui` extension is created. Then it should be assembled with the Python code that controls the interface Widgets' functionalities. To integrate the interface with a given program, we use the `uic` module. This allows us to load the interface through the `(<interface-name>.ui)` method. The function returns a tuple with two elements: the first is a class named as the window defined in the `objectName` in *QtDesigner* Property

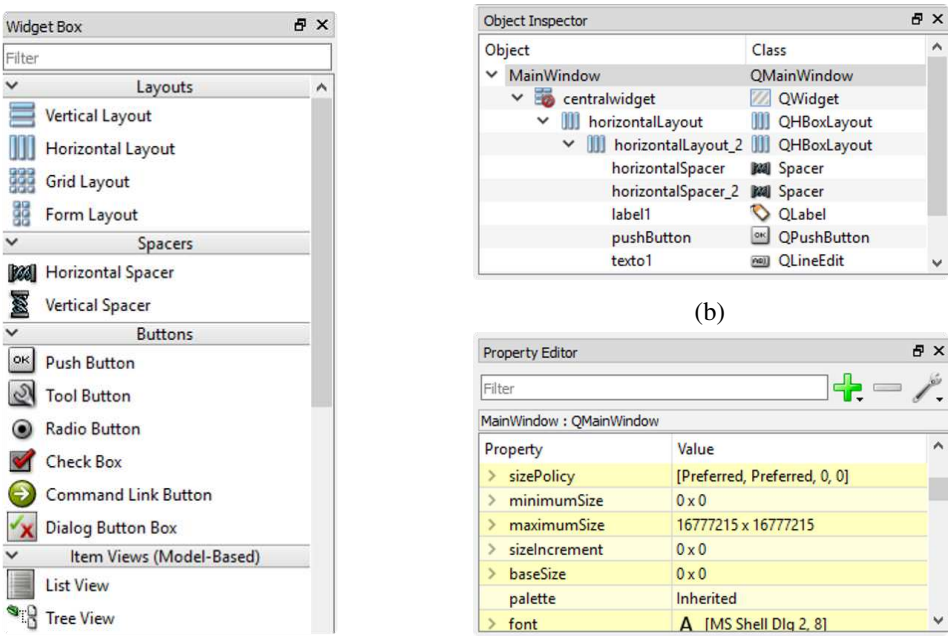


Figure 14.8: This figure shows the main panels available on *QtDesigner*. (a) Shows the Widget Box. (b) Shows the Object Inspector. (c) Shows the Property Editor.

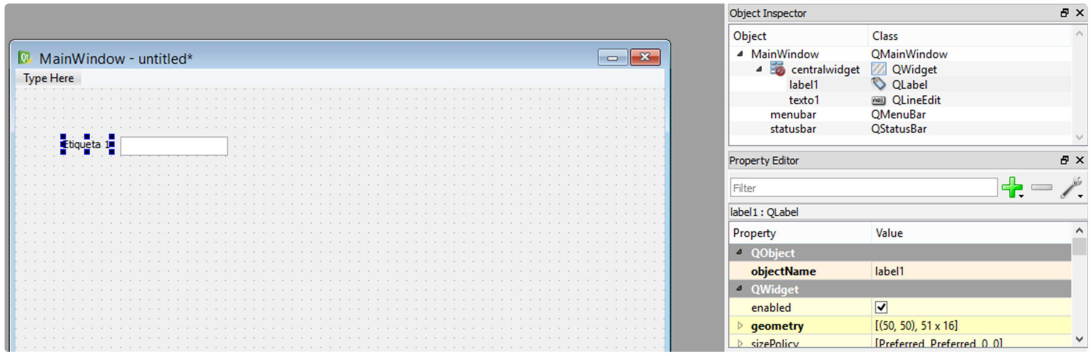


Figure 14.9: An example of how pyqt-designer assigns the object name to a new widget. In this case, the designer adds a new label call `label1`.

Editor; and the second is the name of the class from where it inherits. Figure 14.10 shows where the name appears in the `MainWindow` from the Property Editor.

The following example shows how to perform this procedure with an already created interface. In this case the form variable will include the tuple: `(class 'Ui_MainWindow' class 'PyQt4.QtGui.QMainWindow')`. The **Ui** prefix associated with the name of the class containing the interface is assigned by the `uic` module during interface loading. Once we loaded the interface, it must be initialized within the `__init__()` method, located in the class from where it inherits. This initialization is performed by the `setupUi(self)` method. Applications' creation

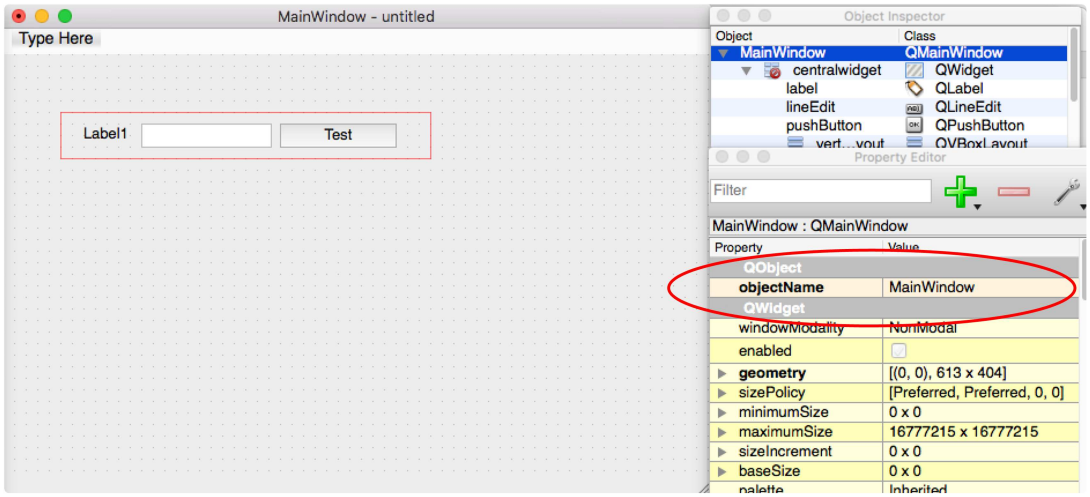
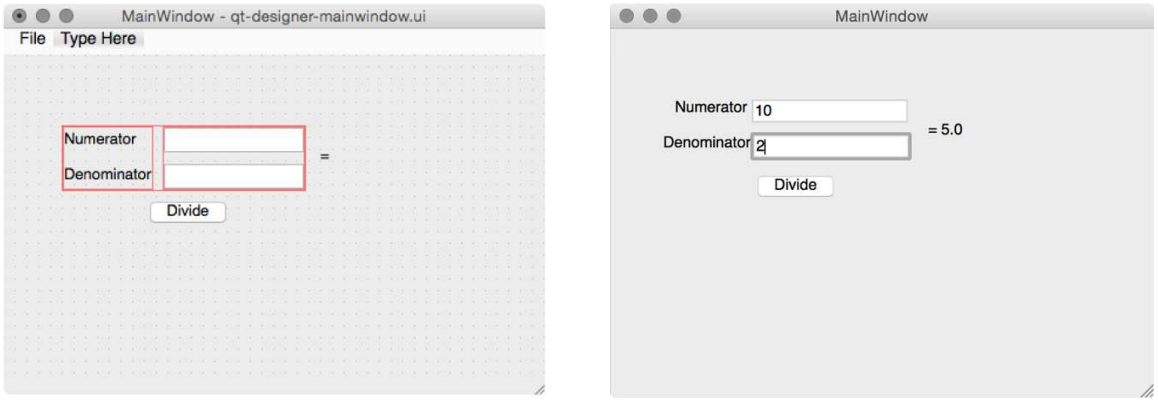


Figure 14.10: We use a red ellipse to point out that the name that Qt Designer assigns to the class representing the window is, in this case `MainWindow`.

must be carried out by using the main program's structure, seen at the beginning of the explanation of graphical interfaces:

```
1  # codes_11.py
2
3  from PyQt4 import QtGui, uic
4
5  form = uic.loadUiType("qt-designer-label.ui")
6
7
8  class MainWindow(form[0], form[1]):
9      def __init__(self):
10         super().__init__()
11         self.setupUi(self)   # Interface is initialized
12
13
14 if __name__ == '__main__':
15     # Application should be initialized just as if it had been
16     # created manually
17     app = QtGui.QApplication([])
18     form = MainWindow()
19     form.show()
20     app.exec_()
```

Each Widget's action must be defined by signals and slots as explained before. Figure 14.11 shows an example of an interface that performs division between two numbers. The button created in *Qt Designer* displays the result on a label.



(a) (b)

Figure 14.11: (a) The design view of the multiplication example. (b) The execution view.

```
# codes_12.py

from PyQt4 import QtGui, uic

form = uic.loadUiType("qt-designer-mainwindow.ui")
print(form[0], form[1])


class MainWindow(form[0], form[1]):
    def __init__(self):
        super().__init__()
        # QtDesigner created interface is initialized
        self.setupUi(self)

        # Button signal is connected
        self.pushButton1.clicked.connect(self.divide)

    def divide(self):
        # This function acts as a slot for de button clicked signal
        self.label_3.setText('= ' + str(
            float(self.lineEdit1.text()) / float(
                self.lineEdit2.text())))
```

```
23
24
25 if __name__ == '__main__':
26     app = QtGui.QApplication([])
27     form = MainWindow()
28     form.show()
29     app.exec_()
```

It is easy to include new Widgets that simplify the user interaction within *Qt Designer*. One example is *radio buttons*, which allow us to capture user options on the form. Figure 14.12 shows a design form using radio buttons and the python code used to verify the values.

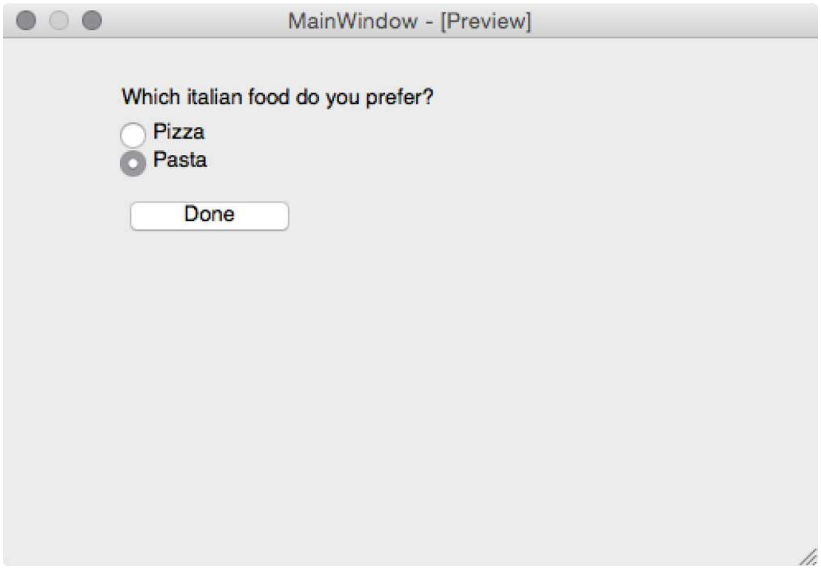


Figure 14.12: Figure shows an execution of a GUI using radio buttons.

```
1  # codes_13.py
2
3  from PyQt4 import QtGui, uic
4
5  form = uic.loadUiType("qt-designer-radiobutton.ui")
6  print(form[0], form[1])
7
8
9  class MainWindow(form[0], form[1]):
10     def __init__(self):
```

```
11          super().__init__()
12          self.setupUi(self)
13
14          self.pushButton1.clicked.connect(self.show_preferences)
15
16      def show_preferences(self):
17          for rb_id in range(1, 3):
18              if getattr(self, 'radioButton' + str(rb_id)).isChecked():
19                  option = getattr(self, 'radioButton' + str(rb_id)).text()
20                  print(option)
21                  self.label2.setText('prefers: {0}'.format(option))
22                  self.label2.resize(self.label2.sizeHint())
23
24
25  if __name__ == '__main__':
26      app = QtGui.QApplication([])
27      form = MainWindow()
28      form.show()
29      app.exec_()
```

Bibliography

[1] `https://docs.python.org/3/library/functions.html`.

[2] `http://www.cs.jhu.edu/~s/musings/pickle.html`.

[3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *ACM SIGPLAN Notices*, volume 31, pages 69–82, 1996.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[5] Narasimha Karumanch. *Data Structure and Algorithmic Thinking with Python*. CareerMonk Publications, 2015.

[6] Dusty Phillips. *Python 3 object oriented programming*. PACKT, 2010.