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
```

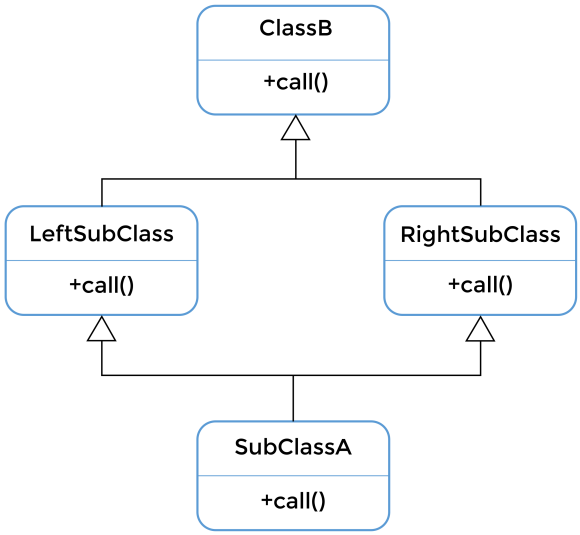

Figure 1.1: The diamond problem

```
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.

```
# mro.py

from diamond_problem_solution import SubClassA

for c in SubClassA.__mro__:
    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:

```
# invalid_structure.py

class X():
    def call_me(self):
        print("I'm X")


class Y():
    def call_me(self):
        print("I'm Y")


class A(X, Y):
    def call_me(self):
        print("I'm A")

```

```
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:

```
1  # polymorfism_1.py
2
3  class Variable:
4      def __init__(self, data):
5          self.data = data
6
7      def representative(self):
8          pass
9
10
11 class Income(Variable):
12     def representative(self):
13         return sum(self.data) / len(self.data)
14
15
16 class City(Variable):
17     # class variable
18     _city_pop_size = {'Shanghai': 24000, 'Sao Paulo': 21300, 'Paris': 10800,
19                       'London': 8600, 'Istambul': 15000,
20                       'Tokyo': 13500, 'Moscow': 12200}
21
22     def representative(self):
23         dict = {City._city_pop_size[c]: c for c in self.data
24                 if c in City._city_pop_size.keys()}
25         return dict[max(dict.keys())]
26
27
28 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.

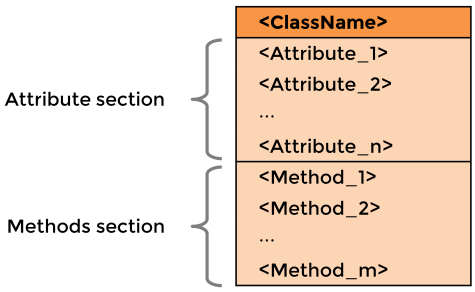

Figure 1.2: UML Class diagram example

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.

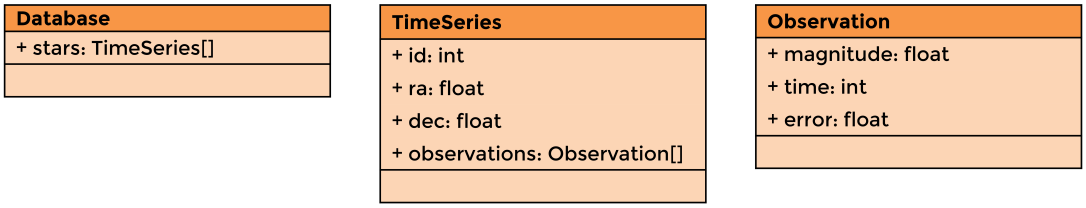

Figure 1.3: UML tables for the stars model

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.

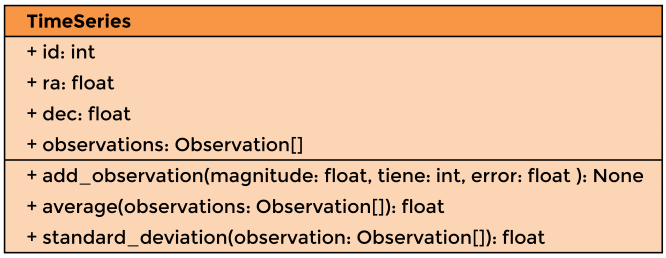

Figure 1.4: UML table for the `TimeSeries` class including the `add_observation`, `average` and `standard_deviation` methods.

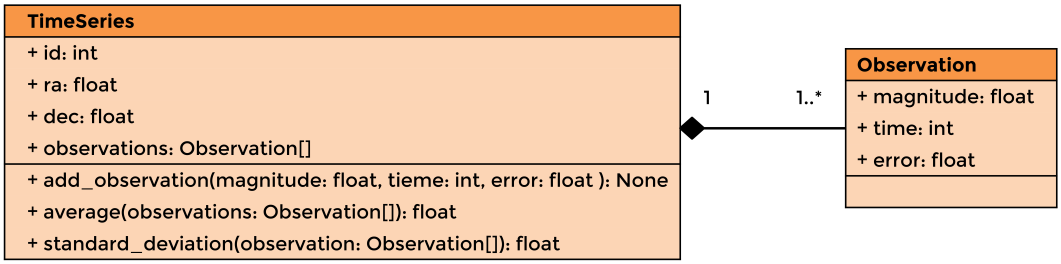

Figure 1.5: Example of the *composition* relation between classes in a UML diagram.

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.

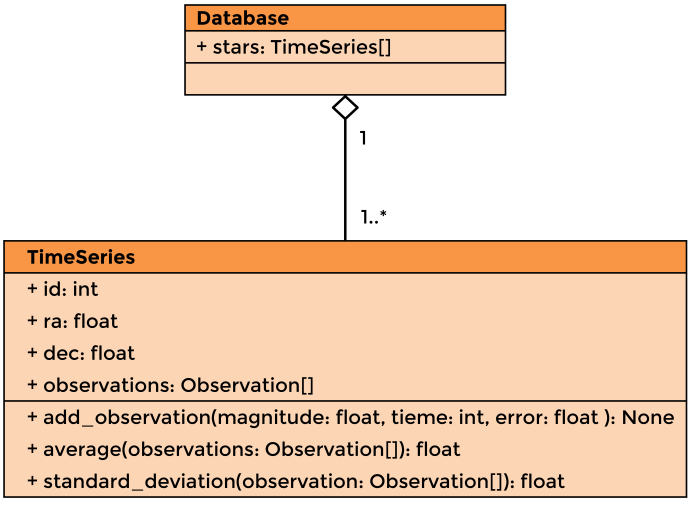

Figure 1.6: Example of the *agreggation* relation between classes in a UML diagram.

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.

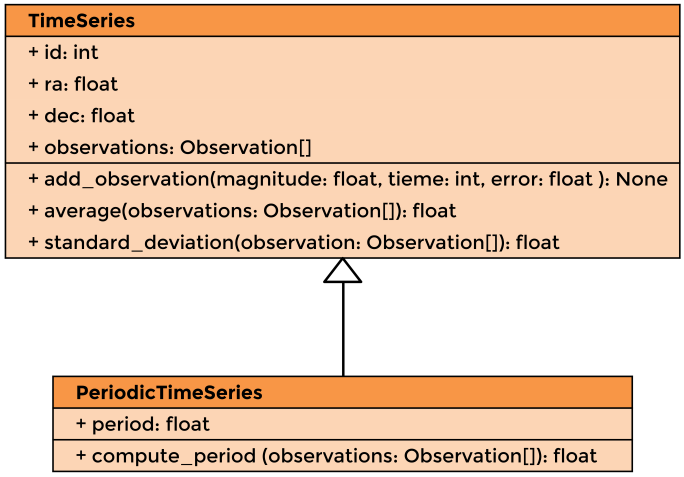

Figure 1.7: Example of the *inheritance* relation between classes in a UML diagram.

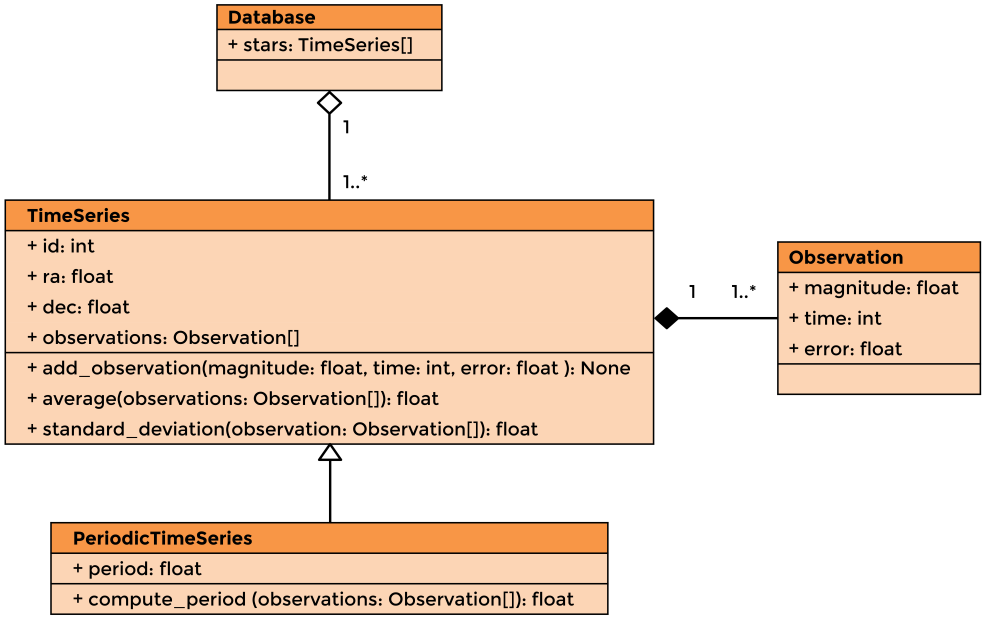

Figure 1.8: The complete UML diagram for the astronomical stars example.

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__`.