FITPED

Python classes

Zenón José Hernández-Figueroa
José Daniel González-Domínguez
Juan Carlos Rodríguez-del-Pino
Viera Michaličková
Jan Přichystal
Kornel Chromiński

www.fitped.eu

2021

Co-funded by the Erasmus+ Programme of the European Union

Work-Based Learning in Future IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Python Classes

Published on

November 2021

Authors

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Viera Michaličková | Constantine the Philosopher University in Nitra, Slovakia

Jan Přichystal | Mendel University in Brno, Czech Republic

Kornel Chromiński | University of Silesia in Katowice, Poland

Reviewers

Jozef Kapusta | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the Erasmus+ Programme of the European Union

ISBN 978-80-558-1785-9

Table of Contents

Classes and Instances

Chapter 1

1.1 Classes and instances

1.1.1

Classes

In Object Oriented Programming, a class is a blueprint to create data objects.

Classes represent entities or concepts while objects are actual instances of those concepts. There can be many objects of the same class.

For example, a chair is a concept that stands for "a type of seat. Its primary features are two pieces of a durable material, attached as back and seat to one another at a 90° or slightly greater angle, with usually the four corners of the horizontal seat attached in turn to four legs" (Wikipedia). The objects in the photos are actual instances of the concept "chair".

1.1.2

Classes represent _____ or concepts while objects are actual _____ of those concepts.

- entities
- instances

🕮 1.1.3

Class definition and instantiation

In Python, a class is created using the reserved word *class* followed by the name of the class:

```
class MyClass:
    """Optional docstring comment"""
    pass
```

The above example creates an empty class named *MyClass*. As the example shows, a class definition could, and should, include an optional docstring comment to document the class.

To instantiate a class we use a constructor expression formed by the name of the class followed by curved brackets, as it were a call to a parameterless function which returns the newly created object:

```
my_object = MyClass()
```

In the above example, *my_object* references an object of type *MyClass*, i.e., an instance of *MyClass*.

📝 1.1.4

Supposing a class named *Square* exists. How can we create a new object of the class *Square*?

- x = new Square()
- x = new Object(Square)
- x = Square()

🕮 1.1.5

Class definition and instantiation (2)

Classes usually include a special function named *__init__* to initialize the objects of the class with some data attributes when they are created.

```
class MyClass:
    """Optional docstring comment"""
    def __init__(self, value1, value2):
        self.attr1 = value1
        self.attr2 = value2
```

In the above example, *MyClass* has an *__init__* function which adds two attributes (*attr1* and *attr2*) when a new object is created. These attributes are initialized with the values passed as the second and the third parameter of *__init__*. The first parameter of *__init__* (*self*) represents the newly created object, to which the attributes are added.

When a class has an *__init__* function, actual values for all parameters of *__init__*, except the first, must be provided to the class in the constructor expression when creating a new object. The first parameter is automatically associated with the new object.

```
o1 = MyClass(1, 2)
o2 = MyClass(3, 4)
```

The above example creates two objects (*o1* and *o2*). The numeric values 1 and 2 are assigned to the attributes *attr1* and *attr2* of *o1*, and the values 3 and 4 are assigned to the attributes *attr1* and *attr2* of *o2*.

1.1.6

See the following class:

```
class Square:
    """a square is a regular quadrilateral (4 equal sides & 4
90° angles)"""
    def __init__(self, side_length):
        self.side_length = side_length
```

How can we created an object of class Square?

- my_square = Square(10)
- my_square = new Square(10)
- my_square = __init__(Square, 10)

1.2 Objects and attributes

1.2.1

Objects' attributes

A class defines a new type of object characterized by a set of attributes that may be data-attributes (variables) or functions (which are known as methods). Data attributes are used to represent the object's state while methods are used to show the object's behaviour by operating on its data attributes to access or modify its state.

For example, if defining a class to represent rectangles we need to establish two attributes: one for the length and one for the width. Possible methods for a rectangle could be one to calculate its area or one to "flip" the rectangle exchanging its length and width.

When instantiating a new object from a class it is initialized with its own instances of the data attributes specified in the class definition. The methods provided in the class declaration can then be used on the new object.

In the case of the rectangle example, each new rectangle will have its own length and width and may be asked to return its area or to "flip".

1.2.2

_____ define new types of objects. An object has a state, represented by a set of _____ (variables), and shows a behavior provided by a set of functions that operate on those variables. Functions that are attributes of an object are known as _____

- data attributes
- Classes
- methods
- instances

1.2.3

Adding attributes dynamically

In Python, attributes can be dynamically added to an object at any time writing the name of the object following by a dot, the name of the attribute and the assignment of a value for the new attribute, as in the following example.

```
o1 = MyClass()
o2 = MyClass()
o1.size = 10
o2.length = 5
```

In the above example, two objects of the same class *MyClass* (*o1* and *o2*) are created. Then the value 10 is assigned to the *size* attribute of *o1* and the value 5 is assigned to the *length* attribute of *o2*. Both attributes are created as a result of these assignations in case they do not exist.

The so-called "dot notation", i.e., <object name>.<attribute name> is used both to add new attributes as to access and modify the yet existing ones.

Dynamic addition of attributes could result in objects of the same class having different attributes and behaviors, as in the previous example. As a blueprint to create objects, a class should provide all attributes that every object of that class must have.

For the case of data attributes, this goal can be achieved by defining an initializer *__init__* method which will be called every time the class is instantiated.

1.2.4

The *__init__* method...

Select one:

- Is called when a new object is created.
- Initializes the class for use.
- Initializes all the data attributes to zero when called.

1.2.5

Adding data attributes at initialization time with *__init__*

Operations used to initialize new objects are usually known as constructors in OOP terminology. In Python, a special method named *__init__* must be added to the class to initialize the objects. This method is called when a new object is created.

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
```

The first parameter (*self* in the above example) represents the object on which the method operates. In the above example, the *__init__* method is used to initialize the newly created object (*self*), adding two data attributes (*self.length* and *self.width*) initialized with the correspondent parameters of *__init__* (*length* and *width*).

When creating an object, values for all the parameters of *__init__* must be passed in the constructor expression, except for the first parameter (self) or any parameter with a default value.

```
my_rect = Rectangle(1.0, 2.0)
```

The __init__ method is not invoked explicitly, but it is called as a result of the creation of the object. The newly created object is automatically assigned to the first parameter, which is not explicitly passed in the constructor expression.

1.2.6

What is true?

- When creating a new object, we must pass the arguments for the __init__ method in the constructor expression
- When instantiating a class, we must do an explicit call to the __init__ method to initialize the new object

1.3 Initialisation, instance attributes (exercises)

1.3.1 Rectangle initializer

Implement, in the file *rectangle.py*, a class named *Rectangle* having an *__init__* method to initialize it. This method must:

1. Accept (apart of the self-argument) two float arguments standing for the length and the width of a rectangle, in this order.
2. Add to the class Rectangle two attribute variables named *length* and *width* and initialize them with the values of the corresponding arguments.

1.3.2 Point initializer

A point on the plane is represented by two coordinates (x, y) relative to some coordinate axes.

Define a class called Point to create objects that represent a point in the plane. The x, y coordinates of the point will be stored as attributes of the object; both attributes will be initialized with the values passed when the object is created so you have to define an initializer with two parameters (apart of self): the first one to initialize x attribute and the second one to initialize y attribute (both parameters are assumed to be int or float numbers).

Example:

```
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x =
3.0, y = 4.5"
```

Example:

```
    p = Point(-8.0, 4)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x = -
8.0, y = 4"
```

1.3.3 Time initializer

Define a class called *Time* with an initialization method. The objects of Time class must have two data attributes called *hour* and *minute* (the *hour* attribute is assumed to be an integer value between 0 and 23, and the *minute attribute* is assumed to be an integer value between 0 and 59). Values for both attributes must be passed as parameters at initialization time (value for the *hour* attribute first).

Example:

```
    t1 = Time(12, 0)
    t2 = Time(8, 54)
```

1.3.4 Person initializer

Define a class called *Person* with an initialization method. The objects of *Person* class must have threes data attributes called *name*, *surname*, and *address*. Values for those attributes must be passed as parameters at initialization time, in the order *name*, *surname*, *address*.

Example:

```
    p1 = Person("Anne", "Ricksaw", "Oxford street, 50. London,
England")
    p2 = Person("Peter", "Forstran", "Cooperhills road, 1023",
Los Angeles, USA)
```

Methods

Chapter 2

2.1 Instance methods

📖 2.1.1

Methods

Objects have methods that operate on their data attributes. Those methods are known as instance methods and are defined within the class to which the object belongs.

```
class Rectangle:
    """A quadrilateral with four right angles"""
    def __init__(self, length, width):
        """Initialize a rectangle with length and width"""
        self.length = length
        self.width = width

    def area(self):
        """Returns the area of the self rectangle"""
        return self.length * self.width
```

In the above example, two methods are defined for the objects of class *Rectangle*: the especial initializer method *__init__*, and a method named *area*.

📝 2.1.2

Which is true?

- Methods are declared within the body of a class
- Methods are declared as instances of a class
- Methods are special initializers of a class

📖 2.1.3

Self

In Python, the methods receive the object on which they operate as their first formal parameter.

```
class Rectangle:
    """A quadrilateral with four right angles"""
    def __init__(self, length, width):
        """Initialize a rectangle with length and width"""
        self.length = length
        self.width = width

    def area(self):
        """Returns the area of the self rectangle"""
        return self.length * self.width
```

In the above example, both methods, *__init__*, and *area*, have a first parameter named *self* that represents the object on which they operate.

The name *self* is used by convention, though Python lets any name for this first parameter.

2.1.4

In this method:

```
    def my_method(self, param1, param2):
        pass
```

- self represents the object on which the method operates
- self represents the class declaring the method
- self represents the method itself

2.1.5

Methods invocation

In Python, methods operating on an object are invoked as attributes of that object using dot notation to bind the method with the object. It is the case for *area* in the below example, where the *area* method returns 50 for *r1* and 24 for *r2* (both, *r1*, and *r2*, had been created just before):

```
r1 = Rectangle(10, 5)
r2 = Rectangle(3, 8)

print(r1.area()) # prints 50
x = r2.area()    # assigns 24 to x
```

The first parameter (the bound object) is omitted when invoking a method, as it is passed implicitly. When an instance method wants to invoke another method of the same object, it uses its first parameter to bind the call.

```
class MyClass:
    ...
    def method1(self, a, b):
        ...
    def method2(self, a, b, c):
        ...
        x = self.method1(a, b)
        ...
```

Invocation of magic methods, as *__init__*, has their own syntax. In the case of *__init__* it is implicitly called just when the object has been created following the constructor expression. Many magic methods are called by built-in functions, although they might also be called directly.

2.1.6

Which is true?

- An instance method requires an object of its class to be created before it can be called
- Instance methods are called as normal functions unless they are associated with an object
- An instance method is bound (using a dot) with the class where it is declared

2.1.7

Magic methods

The *__init__* method, used to initialize objects, belongs to a group of Python special methods known as "magic methods".

```
class Rectangle:
    def __init__(self, length, width):
        """Initialize a rectangle with length and width"""
        self.length = length
        self.width = width
```

Magic methods are used to add special features to classes. There are methods to initialize class instances, methods to get a string representation of an object, methods to overload arithmetic operators to use them with a new class of objects, and more.

Magic methods have predefined names starting and ending with "__" (two underscores), and are invoked by special syntax (note that to create a new object you do not call *__init__*; you write the constructor expression with the arguments needed for *__init__*, and then *__init__* is automatically invoked as part of the internal process to create and initialize the object).

```
r1 = Rectangle(10, 5)
```

The **__*__** syntax is reserved. We must not use it to name our own custom methods because any use is subject to breakage without warning in future versions of Python.

2.1.8

What is true?

- Magic methods are used to add special features to classes
- None of the options is true.
- Magic methods are used to initialize class instances.
- A magic method is any method whose name begins with two leading underscores.

2.2 Instance methods (exercises)

2.2.1 Point coordXY

Objects of the *Point* class represent points in a plane. A point in a plane is identified by two coordinates *(x, y)*, which are represented as attributes of the *Point* class objects.

Add to the Point class an instance method called coordXY that returns a tuple with the pair of coordinates (x, y) of the object on which it is applied.

point.py

```
class Point:
    """A point in a plane"""
    def __init__(self, x, y):
        """A point is initialized with x and y coordinates"""
        self.x = x
        self.y = y

      # Put your code here
```

```
if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y))
```

⌨ 2.2.2 Point move

Objects of the *Point* class represent points in a plane. A point in a plane is identified by two coordinates *(x, y)*, which are represented as attributes of the *Point* class objects.

Add an instance method called *move* to the *Point* class. The *move* method will have two parameters (apart from *self*), which will be used to modify the coordinates (*x*, *y*) of the *self* object (the first parameter will be added to the value of *x* and the second to the value of *y*).

Example:

```
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x = 
3.0, y = 4.5"
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))# Prints "x = 
4.0, y = 3.5"
```

Example:

```
    p = Point(0.0, 0.0)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x = 
0.0, y = 0.0"
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))# Prints "x = 
1.0, y = -1.0"
```

point.py

```
class Point:
    """A point in a plane"""
    def __init__(self, x, y):
        """A point is initialized with x and y coordinates"""
        self.x = x
        self.y = y

    # Put your code here
```

```
if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y))
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))
```

2.2.3 Time increase

Add to the *Time* class an instance method named *increase*. When called, this method increments the value of the minute attribute by one, taking into account the following restrictions:

- when *minute*'s value is equal to 59, if *hour*'s value is less than 23, the *increase* method will increment the *hour* value by one and set the *minute* value to 0, but
- if the *hour*'s value is equal to 23, the increase method will set a 0 for both attributes.

Example:

```
    t = Time(12, 30)
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 12 and minute = 30"
    t.increase()
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 12 and minute = 31"
```

Example:

```
    t = Time(12, 59)
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 12 and minute = 59"
    t.increase()
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 13 and minute = 0"
```

Example:

```
    t = Time(23, 59)
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 23 and minute = 59"
    t.increase()
    print("Hour = {} and minute = {}".format(t.hour, t.minute)
# Prints "Hour = 0 and minute = 0"
```

mytime.py

```
class Time:
    """Represents a time"""
    def __init__(self, h, m):
        """Time objects are initialized with hours and
minutes"""
        self.hour = h
        self.minute = m

    # Put your code here
```

2.3 Representing objects as strings

2.3.1

Formal string representation

The magic method __*repr*__ is called by the repr() built-in function when a formal string representation of an object is required. If possible, this formal representation should look like a valid expression suitable to create an object with the same value.

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value
    def __repr__(self):
        return "Rational({}, {})".format(self.num, self. den)
r1 = Rational(3, 4)
print(repr(r1)) # Prints "Rational(3, 4)"
```

In the above example, the call to the built-in function *repr* triggers a call to the magic method __*repr*__. If __repr__ was not implemented, a default string would be return; this string will look like: "*<Rational object at 0xb7202a4c>*" in which the object class ("Rational") and its location in memory ("*0xb7202a4c*") are identified.

2.3.2

What is the name of the built-in function that is used in Python to request a formal representation of an object as a string?

- repr
- __repr__
- print

2.3.3

Informal string representation

The magic method *__str__* is called when an informal string representation of an object is required.

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value
    def __str__(self):
        return self.num.__str__() + "/" + self.den.__str__()
print(r1) # Prints "3/4"
```

In the above example, the *print(r1)* statement triggers a call to the built-in function *str(r1)*, which in turn calls the method *r1 .__ str __ ()*, if implemented, or else the function *repr(r1)*.

Notice that *__str__* is intended to provide a readable representation of the object while *__repr__* is mainly intended for debugging purposes and so must provide a more informational and unambiguous representation.

2.3.4

Which magic method is intended to provide a more human-readable representation of an object?

- __str__()
- __repr__()

2.4 Objects to string (exercises)

2.4.1 Point to string

Objects of the *Point* class represent points in a plane. A point in a plane is identified by two coordinates *(x, y)*, which in the *Point* class objects are represented as attributes.

Define the magic methods necessary to get both a formal and an informal representation as a string of an object of the Point class.

The formal representation must have the format "Point (x = x, y = y)", like: "Point (x = 3.5, y = 4.0)". The informal representation will have the format "x, y", like: "3.5, 4.0"

point.py
```
class Point:
    """A point in a plane"""
    def __init__(self, x, y):
        """A point is initialized with x and y coordinates"""
        self.x = x
        self.y = y

      # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = Point(3.0, 4.5)
    print("Formal: {}".format(repr(p)))
    print("Informal: {}".format(p))
```

2.4.2 Time to string

Define the magic methods necessary to get both a formal and an informal representation as a string of an object of the Time class.

The formal representation must have the format "Time(h, m)", like: "Time(x = 9, y = 30)". The informal representation will have the format "hh:mm", like: "09:30"

mytime.py
```
class Time:
    """Represents a time"""
    def __init__(self, h, m):
        """Time objects are initialized with hours and
minutes"""
        self.hour = h
        self.minute = m

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    t = Time(12, 5)
    print("Formal: {}".format(repr(t)))
    print("Informal: {}".format(t))
```

2.4.3 Person to string

Define the magic methods necessary to get both a formal and an informal representation as a string of an object of the Person class.

The formal representation must have the format "Person(name, surname)", like "Person("John", "Doe")". The informal representation will have the format "name surname", like "John Doe"

person.py

```
class Person:
    """Represents a person

       Attributes:
       name    : str the name of a person
       surname: str the surname of a person
    """

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    person1 = Person("John", "Doe")

    print("Formal: {}".format(repr(person1)))
    print("Informal: {}".format(person1))
```

Encapsulation

Chapter 3

3.1 Encapsulation (getters, setters)

📖 3.1.1

Classes as an encapsulation mechanism

Encapsulation is usually defined (e.g. Wikipedia) as a construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

In Object Oriented Programming, a class groups a set of data attributes and a set of methods operating on those attributes, so classes are encapsulation mechanisms.

A good encapsulation requires the data attributes of an object being manipulated only by its own methods. To accomplish this in an effective way, those data attributes must be hidden from outside agents.

📝 3.1.2

A class is an encapsulation mechanism because...

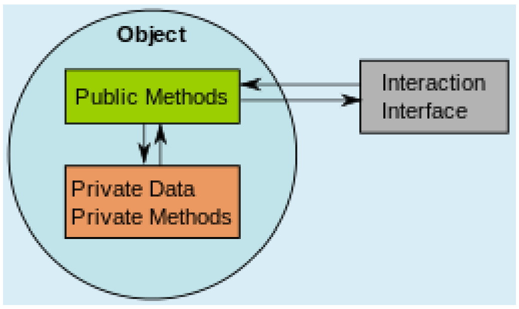
- facilitates the bundling of data with the methods operating on that data
- is a blueprint to create data objects in a program
- provides methods to implement the behavior of the class's objects

📖 3.1.3

Information hiding

Information hiding, being the ability to prevent internal data from being accessed by outer agents, is closely related to encapsulation. Information hiding promotes the principle of separation of public behavior of an object from the details of its implementation while encapsulation provides the mechanism to achieve that principle.

Information hiding identifies two main levels of visibility: public and private; some programming languages provide additional, intermediate levels, for example, protected in Java. Usually, private elements can only be accessed by their owner while public elements can be accessed by everyone. The public elements of a class compound its interface and provide the way to interact with objects of such class.



📝 3.1.4

What is true?

- An interface is composed only of public elements
- Information hiding promotes separation between objects and classes
- Encapsulation is the way to interact with objects

🕮 3.1.5

Hiding attributes in Python

In Python, names starting with two underscores "__", **but not ending** with two underscores, represent a private element that cannot be accessed from outside the class. In the following example, the value attribute is visible (public):

```
class Natural:
    def __init__(self, number):
        self.value = number
n1 = Natural(10)
print(n1.value)
10
```

On the contrary, in the following example, the value attribute is hidden(note the two leading underscores):

```
class Natural:
    def __init__(self, number):
        self.__value = number
n1 = Natural(10)
```

Any attempt to access it from outside raises an error.

```
print(n1.__value)
```

```
Traceback (most recent call last):

File "main.py", line 4, in <module>

    print(n1.__value)
AttributeError: 'Natural' object has no attribute '__value'
```

By convention, the "_" prefix (single underscore) means "stay away", treat this as it were private, though it does not provide a real mechanism to prevent unauthorized access.

3.1.6

Which is true?

- In Python, names that begin, but do not end, with two underscores indicate private elements
- In Python, names that start with two underscores indicate private elements
- In Python, names that end with two underscore indicate private elements

3.1.7

Name Mangling

Python really does not have a true mechanism to make an attribute private and prevent unauthorized access to it. When we name an attribute using two leading underscores and no more than one trailing underscore Python reinforce privacy replacing that name by a mangled named formed by adding at its begin the class name prefixed with a leading underscore. Internally we can use the "normal" unmangled name which is not visible outside the class, but anyone can access the attribute from outside using its mangled name.

```
class Natural:
    def __init__(self, number):
        self.__value = number

n1 = Natural(10)
print(n1._Natural__value) # Correct access using mangled name
print(n1.__value)         # Error: __value is hidden
```

Python deep relies on programmers' responsibility: you can do anything even if you are advised to not but you do it under your responsibility.

The same applies for names beginning with a single underscore: none protection mechanism is applied, but you are warned that an attribute with such sort of is intended name for internal use and can be changed without any warning, so if you use it, it is only your responsibility.

3.1.8

See this code:

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

What is the mangled name for the degrees attribute?

- None
- _Angle__degrees
- __Angle__degrees

3.1.9

Getters and setters

When we hide an attribute, we can provide methods to access it while ensuring encapsulation. The methods that are used to "observe" the value of an attribute are known as "getters", while those that are used to modify them are known as "setters".

In the following example, a getter (*get_value*) and a setter (*set_value*) have been added to the *Natural* class. Also, the *__init__* method has been modified to use the setter instead of making a direct assignment.

```
class Natural:
    def __init__(self, number):
        self.set_value(number)

    def get_value(self):
        return self.__value

    def set_value(self, new_value):
        self.__value = new_value
n1 = Natural(10)
print(n1.get_value())
```

```
n1.set_value(12)
print(n1.get_value())
10

12
```

3.1.10

Which is true?

- A getter is a method which returns the value of a private attribute variable.
- A getter is a method which mutates the state of a private attribute variable.
- A setter is a method which returns the value of a private attribute variable.

3.1.11

Assuming that we provide a getter and a setter for a private variable, we apparently get the same level of access as if it was a public variable.

```
class Natural:
    def __init__(self, number):
        self.set_value(number)

    def get_value(self):
        return self.__value

    def set_value(self, new_value):
        self.__value = new_value
```

The difference is that we can control how its value is modified: a setter is a method used to control changes to a variable. For example, if we want the variable *value* can not take values less than zero, we can modify its setter as in the following example.

```
    def set_value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0
```

In this way, encapsulation prevents an object from entering an erroneous or inconsistent state due to the undue manipulation of its data attributes, this being one of its main advantages.

Another advantage is that, by hiding the implementation, it can be changed without affecting those who are already using the class.

3.1.12

A ... is a method used to control changes to a variable.

3.2 Encapsulation - property, property decorator

3.2.1

Getters and setters syntax drawback

Getters and setters allow controlling the access and modification of private data attributes avoiding erroneous states.

```
class Natural:
    def __init__(self, number):
        self.set_value(number)

    def get_value(self):
        return self.__value

    def set_value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0
```

Using getters and setters requires the access and modification of data attributes to be done by calls to methods instead of directly accessing-and-modifying the variables. This introduces a small syntax drawback. To prevent this complication we will use properties.

```
n1 = Natural(10)
print(n1.get_value()) # instead of print(n1.value)
n1.set_value(-12)     # istead of n1.value = -12
print(n1.get_value()) # instead of print(n1.value)
```

3.2.2

What is true?

- Properties simplify the use of getters and setters
- Properties replace the use of getters and setters
- Properties complicate the use of getters and setters

📖 3.2.3

The property descriptor

To maintain control of access and modification of data attributes and, at the same time, use simple syntax, as if there were no setters and getters, we can define a property, as in the following example:

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

    def get_value(self):
        return self.__value

    def set_value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0

    value = property(get_value, set_value)
```

We first write the setter and getter methods operating on a hidden variable and then use the property descriptor to build a property that can be used as it were a public variable (the getter and the setter will be called automatically when needed):

```
n1 = Natural(10)
print(n1.value)
n1.value = -12
print(n1.value)
```

The property descriptor has four optional arguments:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- *fget* is a function to get the value of the attribute,
- *fset* is a function to set the value of the attribute (if omitted we have a read-only property, else we have a writable one),
- *fdel* is a function to delete the attribute, and
- *doc* is a string providing documentation for the property.

📝 3.2.4

What is true?

- If we omit the setter in a property descriptor, we get a read-only property

- If we omit the getter in a property descriptor, we get a read-only property
- If we omit the setter in a property descriptor, we get a writable property

📖 3.2.5

Hiding the getters and setters

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

    def get_value(self):
        return self.__value

    def set_value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0

    value = property(get_value, set_value)
```

The above example builds a property to avoid the need for explicitly invoking the setter and the getter for a hidden variable, but they could be called anyway because they are public methods.

```
n1 = Natural(10)
print(n1.get_value()) # instead of print(n1.value)
n1.set_value(-12)      # istead of n1.value = -12
print(n1.get_value()) # instead of print(n1.value)
```

Having two way to do the same thing is not a good idea in programming. To avoid this dualism, the getter and the setter must become private methods:

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

    def __get_value(self):
        return self.__value

    def __set_value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0

    value = property(__get_value, __set_value)
```

📝 3.2.6

What is true?

- When we use a property descriptor the getter and the setter should be hidden
- When we use a property descriptor the getter and the setter cannot be hidden
- When we use a property descriptor the getter, but not the setter, should be hidden

🕮 3.2.7

@property decorator

The property descriptor is useful to build a property, but there is an easier way to do that: the @property decorator. We only have to write a getter with the name of the property and decorate it with the *@property* decorator.

```
    @property
    def value(self):
        return self.__value
```

We can then add a setter with the *@<property>.setter* decorator:

```
    @value.setter
    def value(self, new_value):
        self.__value = new_value if new_value >= 0 else 0
```

📝 3.2.8

What is a correct setter decorator for a property named "feature"?

- @feature.setter
- @setter.feature
- @feature.decorator

3.3 Encapsulation (exercises)

3.3.1 Point properties

Objects of the *Point* class represent points in a plane. A point in a plane is identified by two coordinates *(x, y)*, which in the *Point* class objects are represented as attributes.

Make the necessary modifications to the *Point* class to encapsulate the *x* and *y* attributes, converting them into properties with the same names.

Make sure that these properties only support values of type *int* or *float*, so that, if you try to assign them a value of another type, the *TypeError* exception is raised.

> **Note:** The predefined function *type* returns the type of the object that is passed to it as a parameter and can be used to know if the value with which it is intended to initialize any of the properties is of the appropriate type.

point.py
```
class Point:
    """A point in a plane"""
    def __init__(self, x, y):
        """A point is initialized with x and y coordinates"""
        self.x = x
        self.y = y

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y))
```

3.3.2 Time properties

Make the necessary modifications to the *Time* class to encapsulate the *hour* and *minute* attributes, converting them into properties with the same names.

You must ensure the property *hour* only accepts values of type *int* between 0 and 23 and the property *minute* values of type *int* between 0 and 59. If these restrictions are not met, a *ValueError* exception must be thrown.

> **Note:** The predefined function *type* returns the type of the object that is passed to it as a parameter and can be used to know if the value

with which it is intended to initialize any of the properties is of the appropriate type

mytime.py

```
class Time:
    """Represents a time"""
    def __init__(self, h, m):
        """Time objects are initialized with hours and 
minutes"""
        self.hour = h
        self.minute = m

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    t = Time(12, 5)
    print("Formal: {}".format(repr(t)))
    print("Informal: {}".format(t))
```

3.3.3 Person properties

Make the necessary modifications to the *Person* class to encapsulate the *name* and *surname* attributes, converting them into properties with the same names.

You must ensure that both properties only accept non-empty strings as valid values. If these restrictions are not met, a *ValueError* exception must be thrown.

> **Note:** The predefined function *type* returns the type of the object that is passed to it as a parameter and can be used to know if the value with which it is intended to initialize any of the properties is of the appropriate type.

person.py

```
class Person:
    """Represents a person

       Attributes:
       name   : str the name of a person (must have less than 
25 characters)
       surname: str the surname of a person (must have less 
than 40 characters)
    """
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    # Put our code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    person1 = Person("Anne", "Ricksaw")

    print("Name    : {}".format(person1.name))
    print("surname: {}".format(person1.surname))
```

3.3.4 Rectangle area property

Add a new read-only property called *area* to the *Rectangle* class. This property must return the area of the object on which it operates. The area of a rectangle is the product of its length by its width.

rectangle.py

```
class Rectangle:
    """A rectangle is a quadrilateral with four right angles.
       Alternate sides are equal.
    """
    def __init__(self, length, width):
        """Initialization of Rectangle objects
           length: size of the side that we consider the base
of the rectangle
           width: size of a side perpendicular to lenght's
side
        """
        self.length = length
        self.width = width

    @property
    def length(self):
        """Returns lenght of self"""
        return self.__length

    @length.setter
    def length(self, value):
        """Updates length of self"""
        self.__length = value
    @property
    def width(self):
        """Returns width of self"""
        return self.__width

    @width.setter
    def width(self, value):
        """Updates width of self"""
        self.__width = value

    # Put your code here
```

Class Attributes

Chapter 4

4.1 Class variables

📖 4.1.1

Class Attributes

A class can define class attributes that are shared by all the objects in the class, as opposed to the instance attributes, of which each object has its own instance.

For example, suppose we are developing a class to represent money that stores quantities in a base currency (for example, euros) and has methods to return the equivalent amount in a different currency (dollars, yen, pounds, ...).

```
Currency      Exchange
GBP             0.86008
JPY           126.66
USD             1.13190
```

The concrete amount stored is, logically, a different one for each object, but the conversion table that allows translating that amount into different currencies is the same for all and does not need to be stored in each object, it is more appropriate having a single copy, shared by all objects as a class attribute. Class attributes can be both class data attributes (class variables) and class methods.

📝 4.1.2

Which is true?

- A class attribute is shared by all objects in a class
- A class attribute defines the common properties of an object
- A class attribute is replicated in each object of the class

📖 4.1.3

Declaration of class variables

A class variable is declared in the body of the class, outside any method.

```
class MyClass:
    cls_attr = 0

    def __init__(self, value1, value2):
        self.obj_attr1 = value1
        self.obj_attr2 = value2
```

In the above example, *cls_attr* is a class attribute common to all objects of type *MyClass*, while *obj_attr1* and *obj_attr2* are instance attributes and there is a version of each of them for each object of *type MyClass*.

4.1.4

Which is true?

- Class variables are declared outside any method
- Class variables are declared in the __init__ method
- Class variables are declared in the __init_class__ method

4.1.5

Access and modification of class variables

```
class MyClass:
    cls_attr = 0

    def __init__(self, value1, value2):
        self.obj_attr1 = value1
        self.obj_attr2 = value2
```

Any object of the class can access the attributes of the class as if they were of its own...

```
o1 = MyClass(1, 2)
o2 = MyClass(3, 4)
print(o1.cls_attr) # prints 0
print(o2.cls_attr) # prints 0
```

... but only the class can change them.

```
MyClass.cls_attr = 2
print(o1.cls_attr) # prints 2
```

The class itself can obviously access the attribute:

```
print(MyClass.cls_attr) # prints 2
```

But if an object intends to change the value of a class variable, a new instance variable is created for it instead and the object loses the access to the class variable:

```
o2.cls_attr = 3
print(o1.cls_attr)       # prints 2
print(o2.cls_attr)       # prints 3
print(MyClass.cls_attr) # prints 2
```

4.1.6

Which are true?

- Class variables can be accessed but not modified by any object of the class
- Class variables can be accessed and modified by any object of the class
- Class variables can only be accessed by the class itself
- Class variables can only be modified by the class itself

4.2 Class methods

4.2.1

Class methods

A class can have class methods which, as the class variables, are bound to the class itself, not to each object instantiated from it. A class method is distinguished from an instance method because it is declared using the *@classmethod* decorator.

```
class MyClass:
    cls_attr = 0

    def __init__(self, value1, value2):
        self.obj_attr1 = value1
        self.obj_attr2 = value2

    @classmethod
    def sum_cls_attr(cls, value):
        cls.cls_attr += value
```

As for instance methods, the first parameter of a class method has a special meant: it represents the class itself. By convention, this first parameter is usually named *cls*.

Class methods have access to the class variables of their class, and can call other class methods, but cannot access the instance variables or call the instance methods because they are not bound to any instance.

4.2.2

Which decorator is used to declare class methods?

- @classmethod
- @class_method
- @Class_method

4.2.3

Invoking class methods

Class methods are usually invoked as attributes of their class:

```
MyClass.sum_cls_attr(3)
```

They can also be called like attributes of an object, but they do not have any access to the instance attributes of such object:

```
o1.sum_cls_attr(2)
```

A class method can invoke another class method through it class parameter:

```
    @classmethod
    def class_method1(cls, param1, param2):
        ...
    @classmethod
    def class_method2(cls, param1, param2):
        ...
        cls.class_method1(param1, param2)
        ...
```

An instance method can invoke a class method through the attribute __class__ of its self-object:

```
    def instance_method(self):
        self.__class__.class_method1(1, 2)
```

A class method cannot call an instance method because it is not bound to any instance.

4.2.4

See this code:

```
class MyClass:
    @classmethod
    def method1(cls, a, b):
        ...
obj = MyClass()
```

Which of the following are correct ways to call method1?

- MyClass.method1(x, y)
- obj.method1(x, y)
- obj.__class__.method1(x, y)
- MyClass.method1(cls, x, y)

4.2.5

Uses of class methods

A major use of class methods is to manage the class attributes:

```
class MyClass:
    cls_attr = 0
    ...
    @classmethod
    def sum_cls_attr(cls, value):
        cls.cls_attr += value
```

Another important use is like a sort of factory methods to create objects using different sets of arguments for initialization. For example, suppose we have a class to represent angles storing its degrees value. It has a degrees attribute which is initialized with a value passed as a parameter to the *__init__* method:

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

But what is the matter if we have an angle expressed in radians and we want to create an *Angle* object to represent it (of course, internally converted to degrees)? Python classes can have only one *__init__* method. We can use optional parameters to know that the value parameter is expressed in radians and convert it to degrees, but this solution is a bit tricky. A better solution is to use a class method as in the below example:

```
    @classmethod
    def from_radians(cls, value):
        return cls(value * 180.0 / math.pi) # Converts radians
to degrees
```

In the below example both *angle1* and *angle2* represents a 90º angle:

```
angle1 = Angle(90.0)
angle2 = Angle.from_radians(math.pi / 2)
```

4.2.6

When is it a good idea to use a class method to instantiate a class?

- When we need to have different ways to initialize objects
- When we need a way to call the __init__ method from inside the class
- When the __init__ method is unavailable

4.3 Class attributes (exercises)

4.3.1 Rectangle __live_rectangles

Add to the class *Rectangle* a class variable named *__live_rectangles*, initialized with the value 0, and modify the *__init__* method to sum 1 to that variable any time a new rectangle is created.

rectangle.py

```
class Rectangle:
    """A quadrilateral with four right angles"""

    def __init__(self, length, width):
        """Initialize a rectangle with length and width"""
        self.length = length
        self.width = width
```

4.3.2 Currency exchange_rates

The *conver_To* method allows obtaining the equivalent value in another currency according to a pre-established exchange rate. To make the conversion, it is based on a class variable called *exchange_rates*, which is a dictionary that stores, for different currencies, its exchange rate with respect to the euro. This variable has not yet been added to the class.

Add the class variable *exchange_rate* to the Currency class, initializing it according to the following table:

Currency - rate

EUR - 1.0

JPY - 124.83

USD - 1.11918

GBP - 0.85806

currency.py

```
class Currency:
    """Represents a number of euros and
       can give its equivalent value in other currencies
    """

    ######################
    # Put your code here #
    ######################

    def __init__(self, value = 0):
        """"value is an amount of euros """
        self.value = value

    @property
    def value(self):
        """"value is an amount of euros """
        return self.__value

    @value.setter
    def value(self, value):
        """"value is an amount of euros """
        self.__value = value
    def convert_to(self, currency):
        """Returns the equivalent value of self in another
currency
           currency: currency to convert to
        """
        if currency in Currency.exchange_rates:
            return self.value *
Currency.exchange_rates[currency]
        else:
            return None

if __name__ == "__main__":
    # Example of use (not part of the solution)
    c = Currency(10)
    print(c.convert_to("USD"))
    print(c.convert_to("NSN"))
```

4.3.3 Currency rate method

The *conver_To* method allows obtaining the equivalent value in another currency according to a pre-established exchange rate. To make the conversion, it is based on a hidden class variable called *__exchange_rates*, which is a dictionary that stores, for different currencies, its exchange rate with respect to the euro.

Add a class method named *rate* to return the exchange rate for a currency passed as a string parameter. In case that the *__exchange_rates* variable does not contain a key matching with the currency parameter, the *rate* method will return None

Example:

```
Currency   rate
EUR        1.0
JPY      124.83
USD        1.11918
GBP        0.85806
print(Currency.rate("GBP")) # Prints 0.85806
print(Currency.rate("CAD")) # Prints None
```

currency.py

```
class Currency:
    """Represents an amount of euros and
       can give its equivalent value in other currencies
    """
    __exchange_rates = {
        "EUR" : 1.0,
        "JPY" : 124.83,
        "USD" : 1.11918,
        "GBP" : 0.85806
    }

    ######################
    # Put your code here #
    ######################

    def __init__(self, value = 0):
        """"value is an amount of euros """
        self.value = value

    @property
    def value(self):
        """"value is an amount of euros """
        return self.__value

    @value.setter
    def value(self, value):
        """"value is an amount of euros """
        self.__value = value

    def convert_to(self, currency):
        """Returns the equivalent value of self in another 
currency
           currency: currency to convert to
        """
        rate = self.__cls__.rate(currency)
        return rate if rate == None else self.value * rate

if __name__ == "__main__":
    # Example of use (not part of the solution)
    print(Currency.rate("GBP"))
    print(Currency.rate("CAD"))
```

4.3.4 Currency set_rate method

The *conver_To* method allows obtaining the equivalent value in another currency according to a pre-established exchange rate. To make the conversion, it is based on a hidden class variable called *__exchange_rates*, which is a dictionary that stores, for different currencies, its exchange rate with respect to the euro.

Add a class method named *set_rate* to set or update the exchange rate for a currency passed as a string parameter with a value (float) passed as a second parameter.

Example:

```
Currency   rate
EUR        1.0
JPY      124.83
USD        1.11918
GBP        0.85806
print(Currency.rate("GBP")) # Prints 0.85806
print(Currency.rate("CAD")) # Prints None
Currency.set_rate("GBP", 0.839)
Currency.set_rate("CAD", 1.51)
print(Currency.rate("GBP") # Prints 0.839
print(Currency.rate("CAD") # Prints 1.51
```

currency.py

```
class Currency:
    """Represents an amount of euros and
       can give its equivalent value in other currencies
    """
    __exchange_rates = {
        "EUR" : 1.0,
        "JPY" : 124.83,
        "USD" : 1.11918,
        "GBP" : 0.85806
    }

    ######################
    # Put your code here #

    ######################

    @classmethod
    def rate(cls, currency):
        """Returns exchange rate for currency or None"""
        if currency in Currency.__exchange_rates:
            return Currency.__exchange_rates[currency]
        else:
            return None

    def __init__(self, value = 0):
        """"value is an amount of euros """
        self.value = value

    @property
    def value(self):
        """"value is an amount of euros """
        return self.__value

    @value.setter
    def value(self, value):
        """"value is an amount of euros """
        self.__value = value

    def convert_to(self, currency):
        """Returns the equivalent value of self in another
currency
           currency: currency to convert to
        """
        rate = self.__cls__.rate(currency)
        return rate if rate == None else self.value * rate

if __name__ == "__main__":
    # Example of use (not part of the solution)
    print(Currency.rate("GBP"))
    print(Currency.rate("CAD"))
    Currency.set_rate("GBP", 0.839)
    Currency.set_rate("CAD", 1.51)
    print(Currency.rate("GBP"))
    print(Currency.rate("CAD"))
```

Inheritance

Chapter 5

5.1 Inheritance

📖 5.1.1

Inheritance

In Object Oriented Programming, inheritance is a mechanism that allows defining new classes based on existing classes.

New classes defined in such a way are known as derived classes, or subclasses, of the base class (the base class is the class from which a derived class derive; it is also known as the superclass).

Subclasses inherit the features and behaviour from their base class. They usually can override some of those features and behaviour aspects as well as adding new ones. The main benefit of inheritance is the reusability: we do not need to write code to replicate the features of the base class in the derived class.

A real-life example: a bird is a class of animal. Birds have a beak, wings, two legs (features) and generally fly (behaviour). Ostriches, canaries, and penguins are kinds of birds. They belong to classes that are subclasses of the class bird. All of them have beaks, wings and two legs (although very different), but penguins do not fly, they swim as if they were "flying" underwater (behaviour override). Ostriches do not fly either, but they run better than most of the birds, which hardly walk. Canaries can fly, an also they sing very well (and added feature).

📝 5.1.2

What is true?

- Inheritance is a mechanism that allows defining new classes based on existing classes
- Inheritance is a mechanism that allows defining new objects based on existing classes
- Inheritance is a mechanism that allows defining new classes based on existing objects

📖 5.1.3

How to inherit in Python

In Python, to express that a class inherits from another we must write the name of the base class inside curved brackets in the header of the definition of the derived class, next to the name of the new class, like in the following example:

```
class ClassTwo(ClassOne): # ClassTwo inherits from ClassOne
```

Inheritance is transitive, that is, if *ClassTwo* inherits from *ClassOne* and class *ClassThree* inherits from *ClassTwo*, then *ClassThree* inherits indirectly from ClassOne, through *ClassTwo*.

```
class ClassThree(ClassTwo): # ClassThree inherits from
ClassOne, as ClassTwo does
```

An object of type *ClassThree* is also of type *ClassTwo* and of type *ClassOne*.

📝 5.1.4

What is the correct expression to inherit ClassB from ClassA?

- class ClassB(ClassA):
- class ClassB extends ClassA:
- class ClassA(ClassB):

📖 5.1.5

The class object

In Python 3.x, all classes implicitly derive from a common base class named **object**.

```
class MyClass:          # In Python 3.x MyClass inherits from
object
class MyClass(object): # Equivalent declaration in Python 2.x
```

Automatic inheritance from a common base class is usual in many languages and is used to provide support to some basic features which all classes must have.

In Python, the inheritance from the object class provides support for class methods, static methods, properties, defines a Method Resolution Order, and so on.

📝 5.1.6

In Python, from which class derive all the other classes?

- object
- Object
- Class

🕮 5.1.7

Initialization of derived classes

When writing an initialization method for a derived class we have to call the initializer of its superclass. We call it using the prefix "super" to reference the superclass as ClassTwo does in the below example.

```
class ClassOne(object):
    def __init__(self, attr1_value):
        self.attr1 = attr1_value
    ...
class ClassTwo(ClassOne):
    def __init__(self, attr1_value, attr2_value):
        super().__init__(attr1_value)
        self.attr2 = attr2_value
    ...
```

In the above example, ClassOne has an initializer with a parameter which is used to initialize the attr1 attribute. The ClassTwo initializer requires two parameters, one to be passed to the initializer of ClassOne (its superclass) and another to initialize a new attribute. The initializer of a derived class must provide the parameters for the initializer of its superclass, so it must receive those parameters or infer them from the ones that it receives.

📝 5.1.8

See this classes:

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

```
class ClassB(ClassA):
    ...
```

How can we call the initializer of ClassA from the initializer of ClassB?

- class ClassB(ClassA): def __init__(self, x): super().__init__(x) ...
- class ClassB(ClassA): def __init__(self, x): super().__init__(self, x) ...
- class ClassB(ClassA): def __init__(self, x): super(self).__init__(x) ...

5.1.9

Subclass' attributes

An object of a derived class is also an object of the base class from which the subclass derives, so the object has both the attributes defined in the subclass as the ones defined in the superclass. However, it only can access the public attributes of the superclass.

```
class ClassOne:
    def __init__(self, attr1_value):
        self.__attr1 = attr1_value

    def get_attr1(self):
        return self.__attr1

class ClassTwo(ClassOne):
    def __init__(self, attr1_value, attr2_value):
        super().__init__(attr1_value)
        self.__attr2 = attr2_value

    def sum_attributes(self):
        return self.get_attr1() + self.__attr2
```

In the above examples, the method *sum_attributes* of *ClassTwo* can access the method *get_attr1* of *ClassOne*. but the attribute *__attr1* is hidden even though it exists.

5.1.10

Which is true?

- An object of a derived class has all the attributes defined by the superclass

- An object of a derived class does not have access to the attributes defined by the superclass
- An object of a derived class only has the public attributes defined by the superclass

5.1.11

Testing real types of instances

When we have objects of different classes and subclasses...

```
class ClassOne:
    def __init__(self, attr1_value):
        ...
class ClassTwo(ClassOne):
    def __init__(self, attr1_value, attr2_value):
        ...
c1 = ClassOne(1)
c2 = ClassTwo(1, 2)
```

We can check if an object is an instance of a class using the *isinstance function*:

```
print(isinstance(c1, ClassOne)) # Prints True
print(isinstance(c2, ClassOne)) # Prints True
print(isinstance(c1, ClassTwo)) # Prints False
```

We can know the type/class of an object using the *type* function or the *__class__* attribute:

```
print(type(c1))      # Prints <class 'myclass.ClassOne'>
print(c1.__class__) # Prints <class 'myclass.ClassOne'>
```

And we can know if a class if a subclass of another class using the function *issubclass*:

```
print(issubclass(myclass.ClassTwo, myclass.ClassOne)) # Prints
True
```

5.1.12

We can check if an object is an instance of a class using the _____ function, we can know the type/class of an object using the *type* function or the _____ attribute, and we can know if a class if a subclass of another class using the _____ function.

- issubclass
- __class__
- isinstance

5.2 Inheritance (exercises)

5.2.1 Stepped counter

The module *counter* provides a definition for a class named *Counter*. This class has a read-only property named *count* which is assigned a 0 (int) at initialization time. Initialization of Counter objects does not have any additional requirement. *Counter* class also has a method named *add_up* which increments the value of the *count* property by 1 when called.

Example:

```
    c = Counter()
    print(c.count) # Prints 0
    c.add_up()
    print(c.count) # Prints 1
    c.add_up()
    print(c.count) # Prints 2
```

You have to define a new class named *SteppedCounter* which must inherit from *Counter* and match the following requirement:

- Initialization of *SteppedCounter* objects will require, as a parameter, a positive integer value which will be used to initialize a read-only property named *step*.

Example:

```
    c = SteppedCounter(3)
    print(c.count) # Prints 0
    print(c.step)  # Prints 3
    c.add_up()
    print(c.count) # Prints 1
    c.add_up()
    print(c.count) # Prints 2
```

steppedcounter.py

```
from counter import Counter

# Put your code here
```

```
if __name__ == "__main__":
    # Example of use (not part of the solution)
    c = SteppedCounter(3)
    print(c.count)
    print(c.step)
    c.add_up()
    print(c.count)
```

5.2.2 MobilePoint

The module *point* provides a definition for a class named *Point*. Objects of the *Point* class represent points in a plane. A point in a plane is identified by two coordinates *(x, y)*, which are represented as attributes of the *Point* class objects.

You have to define a new class, named *MobilePoint*, which must inherit from Point and add an instance method called *move*. The *move* method will have two parameters (apart from *self*), which will be used to modify the coordinates (*x*, *y*) of the *self* object (the first parameter will be added to the value of *x* and the second to the value of *y*).

Example:

```
    p = MobilePoint(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x = 
3.0, y = 4.5"
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))# Prints "x = 
4.0, y = 3.5"
```

Example:

```
    p = MobilePoint(0.0, 0.0)
    print("(x = {}, y = {})".format(p.x, p.y)) # Prints "x = 
0.0, y = 0.0"
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))# Prints "x = 
1.0, y = -1.0"
```

mobilepoint.py

```
from point import Point

# Put your code here
```

```
if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = MobilePoint(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y))
    p.move(1.0, -1.0)
    print("(x = {}, y = {})".format(p.x, p.y))
```

Polymorphism

Chapter 6

6.1 Polymorphism and inherited methods (methods overriding)

🕮 6.1.1

Polymorphism

Polymorphism is the ability to process objects in a different way, depending on their types, using a single interface. Operators are a typical case of polymorphism, e.g., the same symbol '+' can be used to sum two numbers or concatenate two strings, giving the proper result for each case:

```
x = 5 + 10  # 15
greetings = "Hello " + "world" # "Hello world"
```

In Object Oriented Programming, polymorphism allows treating objects of different subclasses as instances of their superclasses, calling for each case the proper method for the object. This means that functions or methods written to operate on objects of the superclass can also operate on objects of the derived class and treat them accordingly to their classes. For example, suppose these three classes:

```
class BaseClass:
    def message(self):
        return "This is a message from BaseClass";

class ClassOne(BaseClass):
    def message(self):
        return "This is a message from ClassOne";

class ClassTwo(BaseClass):
    def message(self):
        return "This is a message from ClassTwo";
```

And see this function:

```
def showMessage(obj):
    print(obj.message())
```

If we pass to the function an object of any of the three classes, the proper message will be printed:

```
obj1 = BaseClass()
obj2 = ClassOne()
obj3 = ClassTwo()
```

```
showMessage(obj1) # Prints "This is a message from BaseClass"
showMessage(obj2) # Prints "This is a message from ClassOne"
showMessage(obj3) # Prints "This is a message from ClassTwo"
```

Polymorphism deeply relies on method overriding and overloading.

6.1.2

In Object Oriented Programming, what name is given to the ability to process objects in a different way, depending on their types, using a single interface?

6.1.3

Method overriding

Method overriding is a mechanism that allows a subclass to provide its own implementation of a method yet implemented by any of the classes from which it inherits.

In Python, to override a method a class only needs to provide a method with the same name. The method provided in this way hides the one available in the superclass.

```
class BaseClass:
    def message(self):
        return "This is a message from BaseClass"

class ClassOne(BaseClass):
    def message(self):
        return "This is a message from ClassOne"
```

When the *message* method is invoked on an object of type *BaseClass* the one defined in *BaseClass* is run, but when it is invoked on an object of *ClassOne*, the one defined in *ClassOne* is run. The overridden method can be accessed from inside *ClassOne* using the *super* method.

```
    def message(self):
        return super().message() + " and then a message from 
ClassOne"
```

6.1.4

In Python, if a class has a method and we want to override it in a derived class ...

- Just write a method with the same name in the derived class
- You have to write a method with the same name and the decorator @override
- You must access the superclass method using the super() built-in method

6.1.5

Polymorphism and magic methods

In Python, polymorphism is frequently used with magic methods. By nature, a magic method is a special method that a new class should override, writing a new version, to provide the proper behavior for its instances.

New classes usually override the *__init__* method to provide their own initialization or the *__str__* method to provide their own string representation, but any of the magic methods can be overridden if needed.

Any case, when a magic method is overridden, the implementation available for the superclass can be accessed from inside the derived class, if needed, using the *super* method.

```
class ClassOne(object):
    def __init__(self, value):
        self.attr1 = value

class ClassTwo(ClassOne):
    def __init__(self, value_1, value_2):
        super().__init__(value_2)
        self.attr1 = value_1
```

6.1.6

If a class has an initialization method with no parameters other than self and we want to invoke it from the initialization method of a derived class, what would be the correct way to do it?

- super () .__ init __ ()
- super () .__ init __ (self)
- self.super () .__ init ()

6.2 Inheritance and method overriding (exercises)

⌨ 6.2.1 Stepped counter add_up

The module *counter* provides a definition for a class named *Counter*. This class has a read-only property named *count* which is assigned a 0 (int) at initialization time. Initialization of Counter objects does not have any additional requirement. *Counter* class also has a method named *add_up* which increments the value of the *count* property by 1 when called.

Example:

```
    c = Counter()
    print(c.count) # Prints 0
    c.add_up()
    print(c.count) # Prints 1
    c.add_up()
    print(c.count) # Prints 2
```

The module steppedCounter define a new class named *SteppedCounter* which inherits from *Counter* and match the following requirements:

- Initialization of *SteppedCounter* objects will require, as a parameter, a positive integer value which will be used to initialize a read-only property named *step*.

You have to modify *SteppedCounter* to override the *add_up* method of *Counter* in such a way that it will increment the *count* property by the value of step when called.

Example:

```
    c = SteppedCounter(3)
    print(c.count) # Prints 0
    c.add_up()
    print(c.count) # Prints 3
    c.add_up()
    print(c.count) # Prints 6
```

steppedcounter.py

```
from counter import Counter

class SteppedCounter(Counter):
    """A counter with a custom step"""
    def __init__(self, step):
        super().__init__()
        self.__step = step

    @property
    def step(self):
        """Returns current step value"""
        return self.__step

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    c = SteppedCounter(3)
    print(c.count)
    c.add_up()
    print(c.count)
```

6.2.2 timeWithSeconds

The *mytime* module contains the Time class, a class to represent hours and minutes. The initialization of an object of the *Time* class requires two parameters: one for the hours and another for the minutes, in this order. The hours are assumed to be an integer value between 0 and 23 and the minutes an integer value between 0 and 59.

The *Time* class also has defined the *__str__* method. This method returns the hours and minutes represented by an object of the *Time* class using the format "hh:mm".

Example:

```
    c = Time(8, 23)
    print(c) # Prints "08:23"
```

You must define a new class, heir to the *Time* class, called *TimeWithSeconds*. This class has the objective of adding seconds to the information stored by the objects of the *Time* class.

The initialization of an object of the *TimeWithSeconds* class requires three parameters: the hours and minutes, necessary to initialize objects of the Time class, and the seconds, added by the new class.

The *__str__ ()* method must be also overridden so that it returns the information stored using the format "hh: mm: ss".

Example:

```
c = TimeWithSeconds(8, 23, 6)
print(c) # Prints "08:23:06"
```

6.2.3 Point3D

The *point2d* module contains the *Point2d* class, a class to represent points in a plane. The initialization of an object of the *Point2d* class requires two parameters: one for the *x* coordinate and another for the *y* coordinate, in this order.

The *Point2d* class also has a method named *coordinates*. This method returns a tuple with the coordinates (x, y) of the point.

Example:

```
p = Point2d(8, 23)
print(p.coordinates()) # Prints (8, 23)
```

You must define a new class, heir to the *Point2d* class, called *Point3d*. This class will represent points in a tridimensional space.

The initialization of an object of the *Point3d* class requires three parameters: *x* and *y* coordinates, necessary to initialize objects of the *Point2d* class, and the *z* coordinate, added by the new class.

You must also override the *coordinates* method of the *Point2d* class so that it returns a tuple (x, y, z).

Example:

```
p = Point3d(8, 23, 6)
print(p.coordinates()) # Prints (8, 23, 6)
```

6.3 Polymorphism and operators (operators overloading)

6.3.1

Operators overloading

Operators overloading is a type of polymorphism which enables the same operator executes in a different way depending on its arguments. For example, the operator

'+' is used to sum *int* numbers, sum *float* numbers, or concatenate *str* values (strings).

In Python, operators can be overloaded to operate on new classes by defining some specific magic methods. These magic methods can be classified as:

- Comparison magic methods
- Unary operators and functions
- Binary arithmetic operators
- Reflected arithmetic operators

📖 6.3.2

Comparison operators overloading

The following magic methods could be overloaded for comparison:

```
__eq__(self, other) .- defines behavior for the equality
operator ==
__ne__(self, other) .- defines behavior for the inequality
operator !=. If __eq__ is defined, __ne__ is defined
implicitly
__lt__(self, other) .- defines behavior for the less-than
operator <
__gt__(self, other) .- defines behavior for the great-than
operator >
__le__(self, other) .- defines behavior for the less-than or
equal operator <=.
__ge__(self, other) .- defines behavior for the great-than or
equal operator >=.
```

Example:

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value
    def __eq__(self, other):
        if type(other) == Rational:
            return self.num * other.den == self.den *
other.num
        else:
            return False
...
```

```
r1 = Rational(3, 4)
r2 = Rational(6, 8)
print(r1 == r2) # Prints True
```

If *__eq__* had not been defined, the previous example would print false, since it would compare if the two variables reference the same object. Methods __lt__, __le__, __gt__, and *__ge__* are usually known as rich comparison ordering methods.

Note the question in the implementation of the __eq__ method to know if the object with which it is being compared is a *Rational*; Only in that case can you access your *num* and *den* attributes to compare them. In any case, if the other object is not a *Rational*, the result is False (two objects of different types cannot be the same).

6.3.3

defines behavior for the equality operator == _____

defines behavior for the inequality operator != _____

defines behavior for the less-than operator < _____

defines behavior for the greater-than operator > _____

defines behavior for the less-than or equal operator <= _____

defines behavior for the greater-than or equal operator >= _____

- __le__(self, other)
- __lt__(self, other)
- __eq__(self, other)
- __gt__(self, other)
- __ne__(self, other)
- __ge__(self, other)

6.3.4

Total ordering

It is not necessary to implement all of the comparison magic methods to be able to use all the relational operators. If the class is decorated with the decorator *@functools.total_ordering*, it is enough to implement *__eq__* and one of the rich comparison ordering methods *__lt__*, *__le__*, *__gt__*, *__ge__*. The decorator supplies the rest.

Example:

```
from functools import total_ordering

@total_ordering
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value

    def __eq__(self, other):
        if type(other) == Rational:
            return self.num * other.den == self.den *
other.num
        else:
            return False

    def __lt__(self, other):
        return self.num * other.den < self.den * other.num
```

Note that in the implementation of the __lt__ magic method we have not asked about the type of the other object with which it is compared. In the case of equality, we can affirm that, if the objects are not of the same uncle, they cannot be the same but how can we decide which one is smaller? We have chosen to let an error occur in that situation, when trying to use the operation with a data type for which it is not defined.

6.3.5

What decorator must we use to supply total ordering for a class defining *__eq__* and one or more rich comparison ordering methods?

6.3.6

Unary operators and functions

```
__pos__(self) .- implements behavior for the unary operator +
__neg__(self) .- implements behavior for the unary operator -
__abs__(self) .- implements behavior for the abs() function
__invert__(self) .- implements behavior for ~ operator
(bitwise invert)
```

```
__round__(self, n) .- implements behavior for the round()
function
__floor__(self) .- implements behavior for the floor()
function
__ceil__(self) .- implements behavior for the ceil() function
__trunc__(self) .- implements behavior for the trunc()
function
```

Example:

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value

    def __neg__(self):
        return Rational(-self.num, self.den)
...
r1 = Rational(3, 4)
r2 = -r1 # The same as r2 = Rational(-3, 4)
```

6.3.7

What magic method implements behavior for the unary operator -

- __neg__(self)
- __minus__(self)
- __invert__(self)

6.3.8

Binary arithmetic operators

```
__add__(self, other) .- implements behavior for the addition
operator +
__sub__(self, other) .- implements behavior for the
subtraction operator -
__mul__(self, other) .- implements behavior for the
multiplication operator *
__floordiv__(self, other) .- implements for the integer
division operator  //
__div__(self, other) .- implements behavior for the division
operator /
```

```
__mod__(self, other) .- implements behavior for the modulo
operator  %
__divmod__(self, other) .- implements behavior for long
division using the divmod() function
__pow__(self, other[, modulo]) .- implements behavior for the
exponentiation operator **
__lshift__(self, other) .- implements behavior for the left
bitwise shift operator  <<
__rshift__(self, other) .- implements behavior for the right
bitwise shift operator >>
__and__(self, other) .- implements behavior for the bitwise
and operator &
__or__(self, other) .- implements behavior for the bitwise or
operator |
__xor__(self, other) .- implements behavior for the bitwise
xor operator ^
```

Example:

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value
    ...
    def __add__(self, other):
        if type(other) == Rational:
            return Rational(
                self.num * other.den + self.den * other.num,
                self.den * other.den
            )
        elif type(other) == int:
            return self + Rational(other, other)
```

6.3.9

Match each operator with the corresponding magic method:

__add__ _____

__mul__ _____

__sub__ _____

__div__ _____

- +
- -
- /
- *

📖 6.3.10

Reflected binary operators

Magic methods for reflected operators are intended for when using an operator with "swapped operands", that is, *other <op> self* instead of *self <op> other*. They apply when operands are of different types and the left operand does not support the operator (It has not implemented the corresponding magic method).

Magic methods for reflected operators have the same name as "normal" ones, but with a leading 'r', for example, being the addition *__add__(self, other)*, its reflected version is *__radd__(self, other)*.

Example:

```
class Rational:
    def __init__(self, num_value, den_value):
        self.num = num_value
        self.den = den_value
    ...
    def __add__(self, other):
        if type(other) == Rational:
            return Rational(
                self.num * other.den + self.den * other.num,
                self.den * other.den
            )
        elif type(other) == int:
            return self + Rational(other, 1)

    def __radd__(self, other):
        return self + other
```

📝 6.3.11

What is the name of the magic method for the reflected division operator /?

6.4 Operators overloading (exercises)

6.4.1 Point sum

Define the proper magic method to implement behavior for the addition operator +. Sum of two points results in a new point whose coordinates are the sum of the corresponding coordinates of the summed points: (x1, y1) + (x2, y2) = (x1+ x2, y1 + y2)

Example:

```
    p1 = Point(3.0, 4.5)
    p2 = Point(2.5, 3.0)
    p3 = p1 + p2
    print("(x = {}, y = {})".format(p3.x, p3.y)) # Prints "x =
5.5, y = 7.5"
```

point.py

```
class Point:
    """A point in a plane"""
    def __init__(self, x, y):
        """A point is initialized with x and y coordinates"""
        self.x = x
        self.y = y

    # Put your code here

if __name__ == "__main__":
    # Example of use (not part of the solution)
    p1 = Point(3.0, 4.5)
    p2 = Point(2.5, 3.0)
    p3 = p1 + p2
    print("(x = {}, y = {})".format(p3.x, p3.y))
```

6.4.2 Rectangle equality

The *Rectangle* class is designed to create objects that represent rectangles. Add to the *Rectangle* class the definition of the appropriate magic method to implement the behaviour of the equality operator ("="). Two rectangles are considered equal if they have the same lengths and widths.

rectangle.py

```
class Rectangle:
    """A quadrilateral with four right angles"""
    def __init__(self, length, width):
        """Initialize a rectangle with length and width"""
        self.length = length
        self.width = width
```

6.4.3 Time comparable

Make the necessary modifications to the Time class so that the objects of the Time class can be compared using any of the comparison operators (<, ==,! =, <=,>,> =).

Given two objects of the Time class, they are equal if their hour and minute attributes are equal. The one whose hour attribute is less, or equal to the hour attribute, whose minute attribute is less is smaller.

It is recommended to implement the least possible number of magical methods.

mytime.py

```
class Time:
    """Represents a time

       Attributes:
       hour:  int; must be 0 <|= hour <|= 23
       minute : int; must be 0 <|= minute <|= 59
    """

    def __init__(self, hour, minute):
        """
        Parameters:
        hour    : int; value to initialize hour attribute
        minute : int; value to initialize minute attribute
        """
        self.hour = hour
        self.minute = minute

if __name__ == "__main__":
    # Example of use (not part of the solution)
    t1 = Time(12, 30)
    t2 = Time(12, 50)
    print("t1 <| t2? ", t1 <| t2)
    print("t1 = t2? ", t1 == t2)
    print("t1 != t2? ", t1 != t2)
    print("t1 > t2? ", t1 > t2)
    print("t1 <|= t2? ", t1 <|= t2)
    print("t1 >= t2? ", t1 >= t2)
```

6.4.4 Rectangle pow

Define the *__pow__(self, other)* magic method to implement the exponentiation of a *self* raised to the *other-th* power, being *other a* positive *int*.

The result must be a new rectangle whose *length* and *width* are those of self multiplied by the *other* parameter. If the *other* parameter is not *a positive int*, the result will be a new *Rectangle* with the same *length* and *width* than *self*.

rectangle.py

```
class Rectangle:
    """A quadrilateral with four right angles

       Attributes:
       length: float or int
       width: float or int
    """

    def __init__(self, length, width):
        self.length = length
        self.width = width
```

6.4.5 Rectangle multiplier

Define the *__mul__(self, other)* and the *__rmul__ (self, other)* magic methods to implement the multiplication of a Rectangle by an int.

The result must be a new rectangle whose *length* will be that of *self* multiplied by the *other* parameter and whose width will be the same as *self*. If the *other* parameter is not an *int*, the result will be a new *Rectangle* with the same *length* and *width* than *self*.

start.py

```
from rectangle import Rectangle

r1 = Rectangle(1.0, 2.0)
print("length: {}\nwidth: {}".format(r1.length, r1.width))

r2 = r1 * 2
print("length: {}\nwidth: {}".format(r2.length, r2.width))
```

```
r2 = 2 * r1
print("length: {}\nwidth: {}".format(r2.length, r2.width))
```

rectangle.py

```
class Rectangle:
    """A quadrilateral with four right angles

       Attributes:
       length: float or int
       width: float or int
    """

    def __init__(self, length, width):
        self.length = length
        self.width = width
```

6.5 Polymorphism and parameters of a function

📖 6.5.1

Functions (or methods) overloading

Overloading of functions or methods is a type of polymorphism that allows using a function (or method) name with different arguments in the same namespace. To achieve this aim, many programming languages allow writing different functions with the same name but varying the number or types of their arguments. Another way is functions with optional or default parameters.

Python does not allow two functions having the same name in the same namespace but it allows optional parameters. Moreover, Python does not require declaring types for functions arguments, so a function could have different behavior depending on the real types of the arguments passed to it.

In Python, we have a sort of "polymorphic" functions instead of overloaded functions. Below example shows a "Hello world" function with an optional parameter:

```
def hello(name = None):
    if name != None:
        print("Hello " + name + "!")
    else:
        print("Hello!")
```

It can be invoked in two way (with and without parameter):

```
hello()          # Prints "Hello!"
hello("David") # Prints "Hello David!"
```

6.5.2

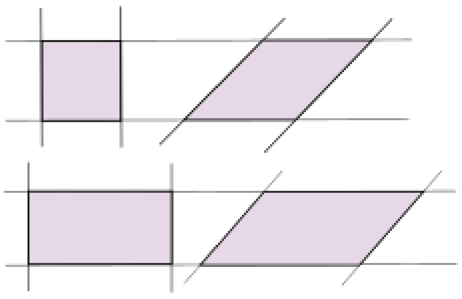
Python allows two functions having the same name in the same namespace as a form of polymorphism.

- False
- True

6.5.3

Polymorphism based on default parameters

The following example shows a class Parallelogram with a polymorphic initializer capable to create any of four possible kinds of parallelograms (square, rectangle, rhombus, or rhomboid). It is supposed that all parameters are int or float numbers.



```
class Paralelogram:
    def __init__(self, side1, angle = 90, side2 = None):
        self.side1 = side1

        if side2 == None or side1 == side2:
            if angle != 90:
                self.kind = "rhombus"
                self.angle = angle
            else:
                self.kind = "square"
        else:
            self.side2 = side2

            if angle != 90:
                self.kind = "rhomboid"
                self.angle = angle
            else:
                self.kind = "rectangle"
```

Below are four different way to instantiate a Parallelogram, with one, two or three arguments:

```
square = Paralelogram(12)
rhombus = Paralelogram(12, 60)
rectangle = Paralelogram(12, side2 = 8)
rhomboid = Paralelogram(12, 60, 8)
```

6.5.4

See the following code:

```
class Paralelogram:
    def __init__(self, side1, angle = 90, side2 = None):
        self.side1 = side1

        if side2 == None or side1 == side2:
            if angle != 90:
                self.kind = "rhombus"
                self.angle = angle
            else:
                self.kind = "square"
        else:
            self.side2 = side2

            if angle != 90:
                self.kind = "rhomboid"
                self.angle = angle
            else:
                self.kind = "rectangle"
```

What value will print the following command:

```
print(Paralelogram(8, 90, 8).kind)
```

- square
- rectangle
- rhombus
- rhomboid

6.5.5

Polymorphism based on types of parameters

The following example shows a function that returns a fraction number or a complex number, depending on if its arguments are of type int or of type float.

```
from fractions import Fraction

def numberFunc(number1, number2):
    if type(number1) == int and type(number2) == int:
        return Fraction(number1, number2)
    elif type(number1) == float or type(number2) == float:
        return complex(number1, number2)
```

Below are three forms of calling the function:

```
print(numberFunc(1, 2))      # Prints "1/2"
print(numberFunc(1.0, 2.0)) # Prints "(1+2j)"
print(numberFunc("1", "2")) # Prints "None"
```

6.5.6

See the following code:

```
from fractions import Fraction

def numberFunc(number1, number2):
    if type(number1) == int and type(number2) == int:
        return Fraction(number1, number2)
    elif type(number1) == float and type(number2) == float:
        return complex(number1, number2)
```

What value will print the following command:

```
print(numberFunc(1, 2.0))
```

- None
- 1/2
- (1+2j)

Immutable Classes

Chapter 7

7.1 Immutable classes

📖 7.1.1

Immutable classes

An immutable class is designed to create immutable objects. An immutable object is an object which cannot be modified after its creation. Numbers, strings, and tuples are examples of Python's built-in immutable objects. Lists and dictionaries are examples of mutable objects. In Python, objects created by custom classes are mutable by default.

```
class Mutable:
    def __init__(self, value1, value2):
      self.attr1 = value1
      self.attr2 = value2
```

If an object is mutable, we can change the values of its attributes:

```
mutable_object = Mutable(1, 2) # An object is created with
attr1 = 1 and attr2 = 2
mutable_object.attr1 = 3    # Value of attr1 changes to 3
```

Also, new attributes can be added to the object:

```
mutable_object.attr3 = 4  # attr3 is added to the object with
value 4
```

Attributes can be removed too:

```
delattr(mutable_object, "attr2") # The object no longer has
the attribute attr2
```

📝 7.1.2

What is true?

- An immutable class creates immutable objects
- An immutable class cannot be inherited
- An immutable class is a sort of tuple

📖 7.1.3

Immutable using *__setattr__*

If we want to avoid the modification of object attributes, we can override the *__setattr__* magic method. This method is automatically called when we attempt to assign an attribute.

```
class Immutable:
    ...
    def __setattr__(self, key, value):
        raise AttributeError("Immutable can not be modified")
```

With this implementation, an error is raised every time assigning an attribute is attempted. The problem is that the error is raised even if the assignment is done in the initializer itself.

```
class Immutable:
    def __init__(self, value1, value2):
        self.attr1 = value1
        self.attr2 = value2
    ...
```

So we have to modify the initializer to call the *__setattr__* of the superclass, bypassing the local one.

```
class Immutable:
    def __init__(self, value1, value2):
        super().__setattr__("attr1", value1)
        super().__setattr__("attr2", value2)
    ...
```

📝 7.1.4

What is true?

- __setattr__ is called when an attribute assignment is attempted
- __setattr__ is called by __init__ when a class is instantiated
- __setattr__ is called every time that an object is created

📖 7.1.5

Using *__slots__*.

Override of *__setattr__* avoids the "normal" modification of attributes by assignment.

```
immutable_object = Immutable(1, 2) # An object is created with
attr1 = 1 and attr2 = 2
immutable_object.attr1 = 3         # Attempt to modify attr1
Traceback (most recent call last):

  File "main.py", line 4, in <module>

    immutable_object.attr1 = 3

  File "/home/p16582/immutable.py", line 9, in __setattr__

    raise AttributeError("Immutable can not be modified")

AttributeError: Immutable can not be modified
```

But we can yet modify or add attributes using the *__dict__* attribute of the object. It is a default attribute (a dictionary) which stores all the writable attributes of the object.

```
immutable_object.__dict__["attr1"] = 4
immutable_object.__dict__["attr3"] = 5
```

To avoid this shortcoming, we can use *__slots__*, a namedtuple intended to improve the performance of objects: using *__slots__* deny the creation of *__dict__* so saving space and improve speed.

```
class Immutable:
    __slots__=("attr1", "attr2")
    ...
```

Now we no longer can't access *__dict__*.

```
immutable_object.__dict__["attr1"] = 4 # Attempt to modify
attr1
Traceback (most recent call last):

  File "main.py", line 7, in <module>

```

```
    immutable_object.__dict__["attr1"] = 4

AttributeError: 'Immutable' object has no attribute '__dict__'
```

7.1.6

Which are true?

- When using __slots__, objects cannot be assigned new variables not listed in the __slots__ definition
- Variables listed in the __slots__ definition can be assigned if __setAttr__ does not prevent it
- When using __slots__, we only can add new attributes using the __dict__attribute

7.1.7

Avoiding deletion of attributes.

We can override the __*setattr*__ method to avoid modification of attributes, but they can yet been deleted if we don't override the __*delattr*__ method too.

```
class Immutable:
    ...
    def __delattr__(self, key):
        raise AttributeError(f"Attribute {key} can not be 
deleted")
```

When we do that, any attempt to delete an attribute will raise an error.

```
delattr(immutable_object, "attr1")
Traceback (most recent call last):
  File "main.py", line 11, in <module>

    delattr(immutable_object, "attr1")

  File "/home/p18545/immutable.py", line 11, in __delattr__

    raise AttributeError("Attribute {} can not be deleted".for
mat(key))
AttributeError: Attribute attr1 can not be deleted
```

7.1.8

What is true?

- __delattr__ is called when an attribute is attempted to be removed
- __delattr__ is called when an object overrides a superclass' attribute
- __delattr__ is called when we want to avoid deletion of an attribute

7.1.9

Immutable classes inheriting from *tuple*

Another way to get an immutable class is inheriting from *tuple*. Being tuple immutable, any class inheriting from *tuple* will be immutable.

```
class Immutable(tuple):
    def __new__(cls, value1, value2):
        return tuple.__new__(cls, (value1, value2))

    @property
    def attr1(self):
        return self[0]

    @property
    def attr2(self):
        return self[1]
```

Notice:

- The *__new__* magic method is used to create a tuple which stores the values for the attributes as elements. In Python, *__new__* is called at the first stage of instantiation, it receives the class and the arguments passed to the constructor expression and uses them to prepare and return the new object which is then passed to the *__init__* method together with those arguments. The *__new__* method is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.
- Read-only properties are used to provides a way to access the stored values by name, as usual for attributes.

A minor drawback of this implementation is that values can be accessed by index, as elements of a tuple. Is this is a problem, we can override the __getitem__ magic method, which is intended to implement the access by index to an object.

```
immutable_object = Immutable(1, 2)
immutable_object.attr2 == immutable_object[1]
```

A major drawback is that we can add elements to __dict__, although we cannot use them as regular attributes.

```
immutable_object.__dict__["attr1"] = 4
print(immutable_object.__dict__["attr1"]) # Prints 4
print(immutable_object.attr1)             # Prints 1
```

7.1.10

What method must we override to avoid access by index when inheriting an immutable class from tuple?

- __getitem__
- __getelement__
- __itemAt__

7.1.11

Immutable classes using *namedtuple*

The simplest way to get an immutable class is using the factory function *namedtuple*, intended to create tuples with field names. The field names are passed as a sequence of strings or a string with field names separated by whitespaces and/or commas.

```
from collections import namedtuple

class Immutable(namedtuple("Immutable", "attr1, attr2")):
    ...
```

Unlike when inheriting from *tuple*, we don't need to override the *__new__* method nor define any properties. By the other hand, access by index continues being possible as well as inserting elements into the *__dict__* attribute, although we cannot use them as regular attributes. Instances are created in the usual way:

```
immutable_object = Immutable(1, 2)
```

7.1.12

What is true?

- Inheriting from a namedtuple simplifies the definition of immutable classes
- Inheriting from a namedtuple prevents access to the attributes by index

- Inheriting from a namedtuple prevents access to the __dict__ attribute

7.2 Immutability (exercises)

7.2.1 Rectangle immutable

Modify the implementation of the *Rectangle* class to convert it to an immutable class inheriting from *tuple* (preferably using *namedtuple)*.

rectangle.py

```
class Rectangle:
    """A rectangle is a quadrilateral with four right angles.
       Alternate sides are equal.
    """
    def __init__(self, length, width):
        """Initialization of Rectangle objects
           length: size of the larger sides
           width: size of the shorter sides
        """
        self.length = length
        self.width = width

    def area(self):
        """The area of a rectangle is the product of its
length by its width"""
        return self.length * self.width

if __name__ == "__main__":
    # Example of use (not part of the solution)
    r = Rectangle(12, 8)
    print(r.length) # Prints 12
    print(r.width)  # Prints 8
    print(r.area()) # Prints 96
```

7.2.2 Point immutable

Modify the implementation of the *Point* class to convert it to an immutable class inheriting from *tuple* (preferably using *namedtuple)*.

point.py

```
class Point:
    """A point in a plane

        Attributes:
        x: float, x coordinate
        y: float, y coordinate
    """

    def __init__(self, x, y):
        self.x = x
        self.y = y

if __name__ == "__main__":
    # Example of use (not part of the solution)
    p = Point(3.0, 4.5)
    print("(x = {}, y = {})".format(p.x, p.y))
```

7.2.3 Time immutable

Modify the implementation of the *Time* class to convert it to an immutable class inheriting from *tuple* (preferably using *namedtuple)*.

mytime.py

```
class Time:
    """Represents a time"""

    def __init__(self, hour, minute):
        """Time objects are initialized with hours and 
minutes"""
        self.__hour = hour
        self.__minute = minute

    # Hour an minute are readonly properties

    @property
    def hour(self):
        return self.__hour

    @property
    def minute(self):
        return self.__minute
```

7.2.4 Rectangle immutable II.

Modify the implementation of the *Rectangle* class to convert it to an immutable class **not inheriting** from *tuple*

rectangle.py

```
class Rectangle:
    """A rectangle is a quadrilateral with four right angles.
       Alternate sides are equal.
    """
    def __init__(self, length, width):
        """Initialization of Rectangle objects
           length: size of the larger sides
           width: size of the shorter sides
        """
        self.length = length
        self.width = width

    def area(self):
        """The area of a rectangle is the product of its
length by its width"""
        return self.length * self.width

if __name__ == "__main__":
    # Example of use (not part of the solution)
    r = Rectangle(12, 8)
    print(r.length) # Prints 12
    print(r.width)  # Prints 8
    print(r.area()) # Prints 96
```

Objects

Chapter 8

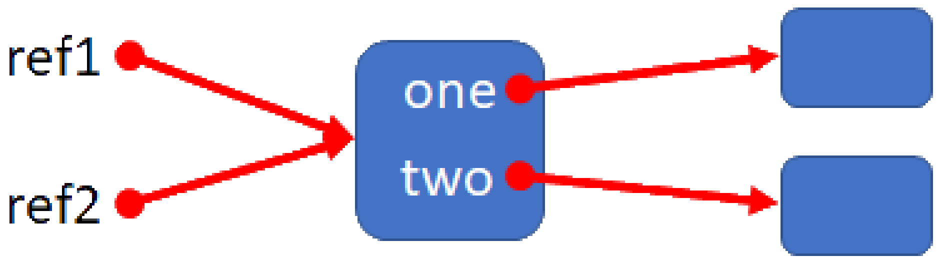
8.1 Copying objects

📖 8.1.1

Assignment

In Python, variables are references to objects. When we assign a variable to another one, the assignment operator "=" copies a reference, so both variables will reference the same object.

```
class One:
    pass

class Two:
    pass

class Three:
    def __init__(self, one, two):
        self.one = one
        self.two = two

ref1 = Three(One(), Two())
ref2 = ref1
```



When managing mutable objects, we may need to get an independent copy of an object which can be mutated without change the original one. Python provides a module named *copy* for this purpose. This module allows two types of copies: shallow and deep.

📝 8.1.2

When is more useful to make a copy of an object?

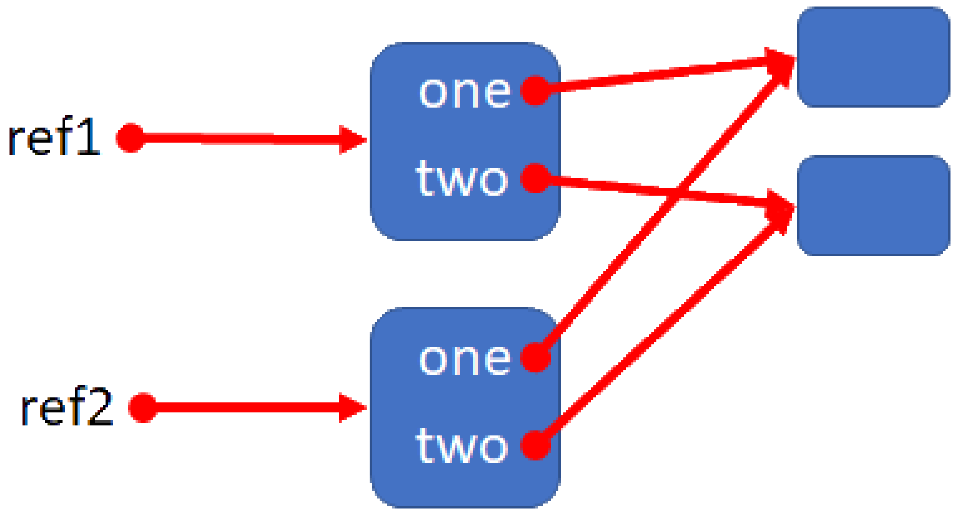
- When we are managing mutable objects
- When we are managing immutable objects
- Is equally useful for both mutable or immutable objects

📖 8.1.3

Shallow copy

We can do a shallow copy of an object using the copy function from the copy module.

```
import copy

class One:
    pass

class Two:
    pass

class Three:
    def __init__(self, one, two):
        self.one = one
        self.two = two

ref1 = Three(One(), Two())
ref2 = copy.copy(ref1)
```



When an object has nested references to other objects, shallow copying makes a copy of the root object and then assigns the nested references, so sharing the objects referenced by them.

📝 8.1.4

A shallow copy does not copy nested objects.

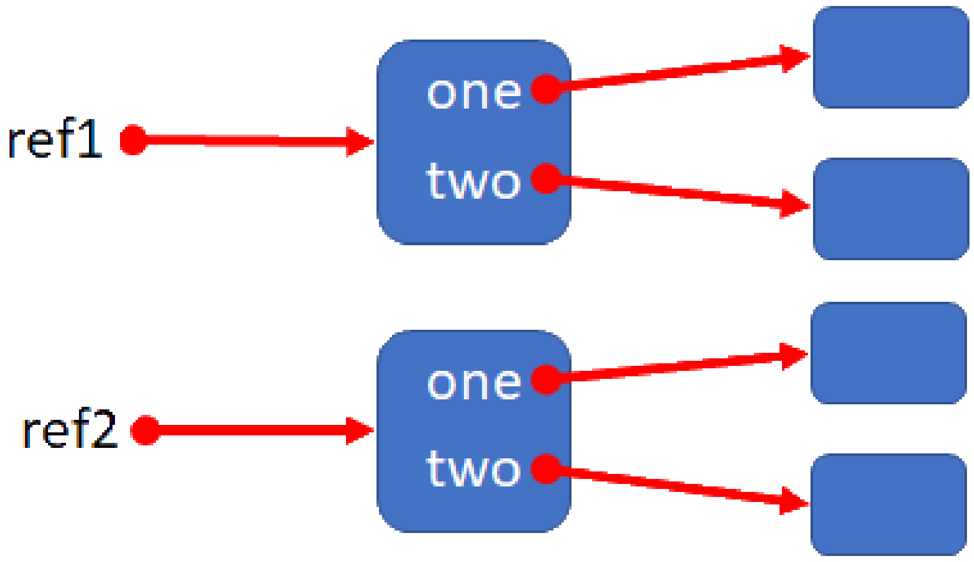
- True
- False

📖 8.1.5

Deep copy

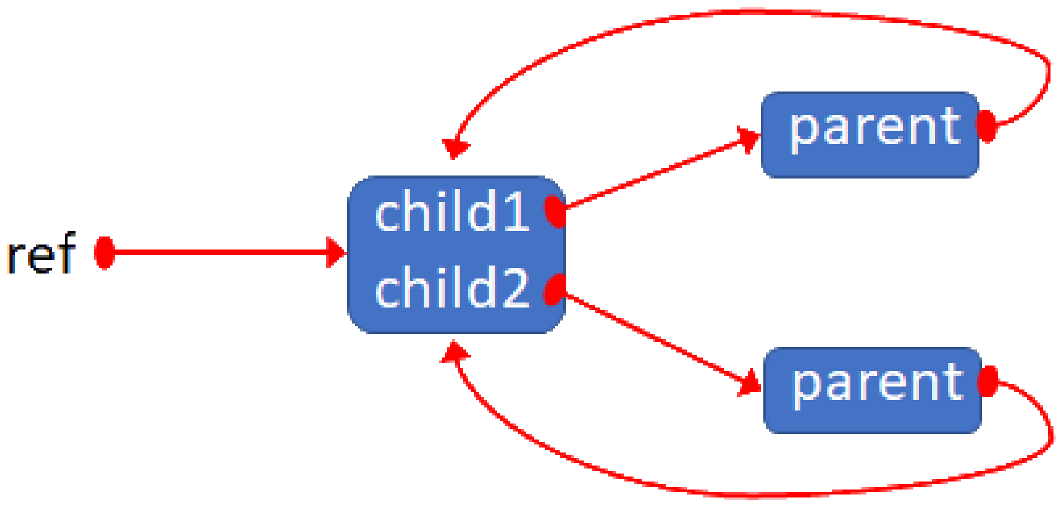
We can get a deep copy of an object using the *deepcopy* function from the copy module.

```
import copy

class One:
    pass

class Two:
    pass

class Three:
    def __init__(self, one, two):
        self.one = one
        self.two = two

ref1 = Three(One(), Two())
ref2 = copy.deepcopy(ref1)
```



When an object has nested references to other objects, deep copying makes a copy of the root object and then recursively deep copies the nested objects.

A deep copy may cause a recursive loop when coping compound objects that, directly or indirectly, contain references to themselves. To prevent this, *deepcopy* keeps a dictionary of objects already copied.



8.1.6

Why deep copy keeps a dictionary of objects already copied?

- To prevent recursive loops
- To prevent self-references
- To prevent duplication of shared data

8.1.7

Magic methods for copying

Python allows customizing both shallow copy and deep copy by defining the corresponding magic methods.

To customize the shallow copy, we have to define the *__copy__* magic method. This method does not require any additional parameter (only the reference to the object on which it operates).

To customize the deep copy, we have to define de *__deepcopy__* magic method. This method requires an additional parameter: the dictionary to remember objects already copied.

Customization of deep copy can be useful to prevent the copy of data that are intended to be shared, so saving memory.

```
import copy

class Child:
    def __init__(self, value, parent):
        self.value = value
        self.parent = parent

class Parent:
    def __init__(self, self_value = None, child_value = None):
        if self_value != None:
            self.value = self_value # This value is to be
shared between copies
        if child_value != None:
            self.child = Child(child_value, self) # This makes
a circular reference

    def __deepcopy__(self, memo):
        if not str(id(self)) in memo: # Object must be copied
            copy_object = Parent() # A new object to be the
copy of self
            memo[str(id(self))] = copy_object # Register
copy_object as copy of self
            if self.value != None:
                copy_object.value = self.value # This value is
shared
            if self.child != None:
                # child attribute is deep copied
                copy_object.child = copy.deepcopy(self.child,
memo)
            return copy_object
        else: # Object has been copied yet, we return the copy
            return memo[str(id(self))]

ref1 = Parent(1, [2])
ref2 = copy.deepcopy(ref1)
```

Above example shows how to define *__deepcopy__* to prevent duplication of data intended to be shared (*value attribute* of the *Parent* object). The *memo argument* is the dictionary to avoid recursive looping, note how we use it to decide if the *Parent* object must be copied. *Child* objects use the default deepcopy.

8.1.8

Which method has more parameters?

- __deepcopy__
- __copy__

8.2 Destroying objects

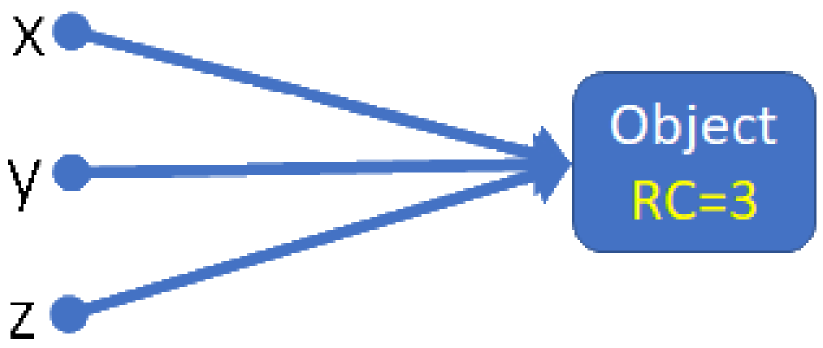
8.2.1

Reference counting garbage collection

Python uses a reference counting garbage collection algorithm. This means that for each object there is a counter of the number of references pointing to it. This counter increases when a new reference is set and decreases when a reference disappears.

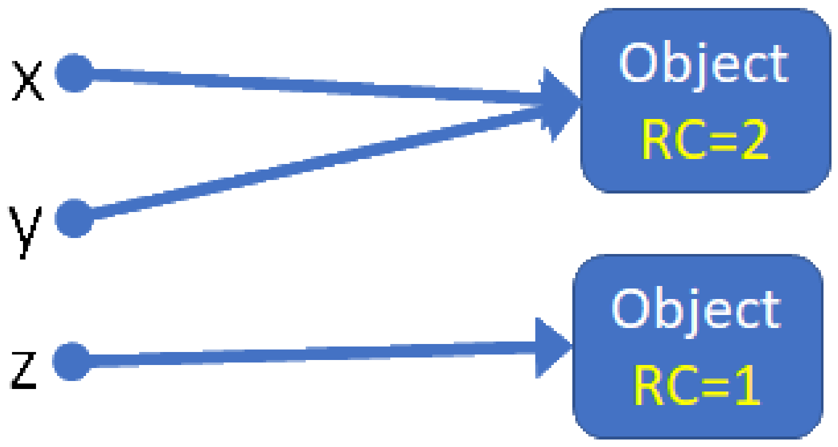
A reference is counted when a variable is assigned the object.

```
x = MyClass()

y = x

z = x
```



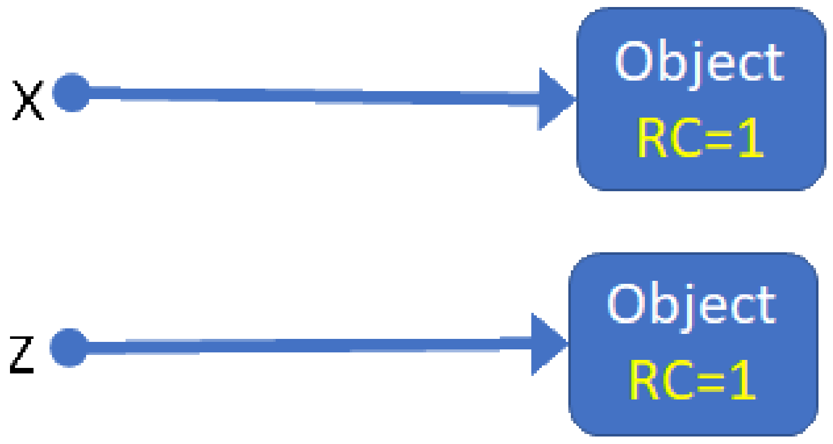
A reference is discounted when a variable is assigned a different object.

```
z = MyClass()
```



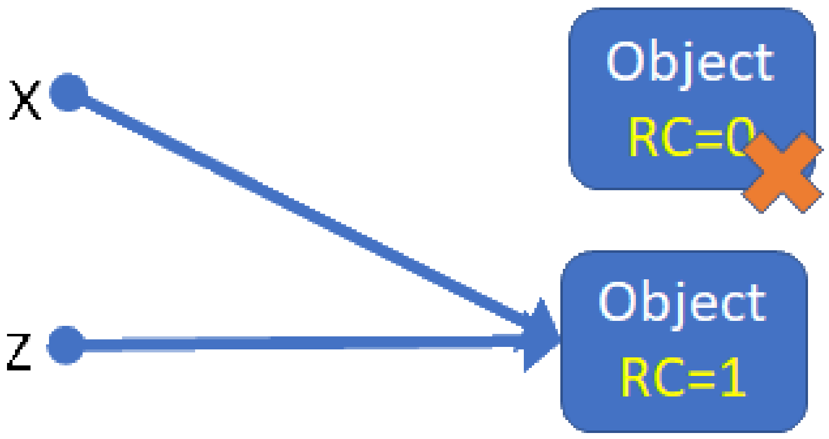
A reference is discounted too when the *del* built-in function is invoked. This function removes a name and decrements the reference counter of the object referenced by it, but does not remove the object itself.

```
del(y)
```



When the reference counter of the object reaches the zero value, the object can be removed by the garbage collector, but there are no guarantees about the exact moment this removing will occur.

```
x = z
```



8.2.2

Which is true?

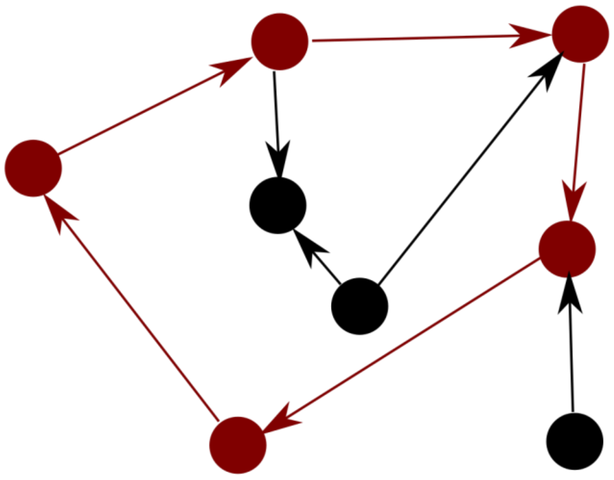
- Objects in Python have a reference counter
- The built-in function del() removes the object referenced by its parameter
- Calling the built-in function del triggers a call to the garbage collector

8.2.3

The __*del*__ magic method

The __*del*__ magic method is called when an object is about to be destroyed by the garbage collector, which happens at some (undefined) point after all references to the object have been deleted. It provides an opportunity to free resources that are not under the garbage collector control; for example, it could be used to commit and close a database connection previously opened and used by other methods during the object's lifetime.

We must take into account some issues:



- We do not know when the *__del__* method is going to be called. Call, if any, will occur at any moment after the reference counter of the object reaches the zero value.
- It could even never be called. If there are circular references among objects and those objects have a *__del__* method, then they can not be removed by the garbage collector.
- Any exception risen in the *__del__* method is ignored.

Knowing these issues, we can use the __del__ method in a safe way if we avoid circular references and don't worry about the exact moment of its execution.

8.2.4

Which are true?

- The __del__ method sometimes is not called
- The __del__ method is called by the built-in function del
- The __del__ method is called just when the reference counter reaches zero
- The __del__ method is called after the reference counter reaches zero

More Inheritance

Chapter 9

9.1 More about inheritance: multiple inheritance, abstract classes

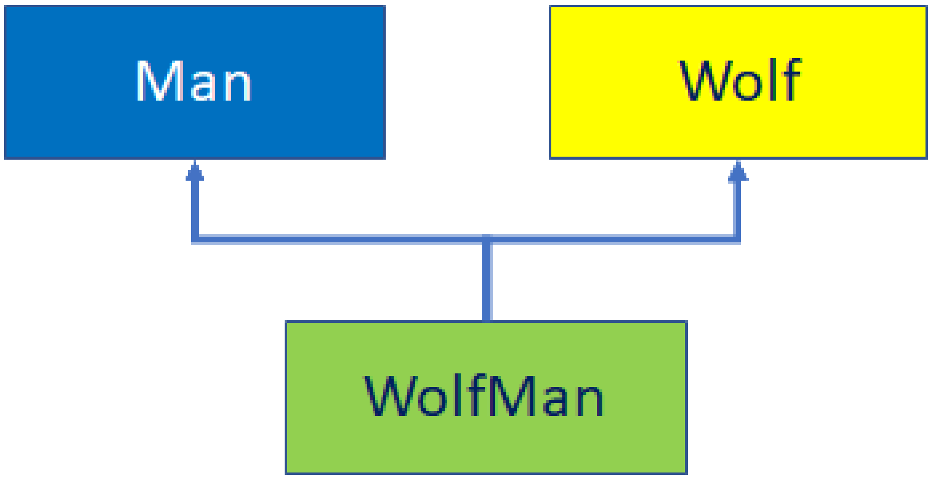
9.1.1

Multiple inheritance

A class can inherit from several classes simultaneously:

```
class WolfMan(Man, Wolf):
    pass
```

WolfMan class objects would inherit the properties and behavior of both the *Man* class and the *Wolf* class, in addition to adding their own.



9.1.2

What is multiple inheritance?

- A class can inherit from several different classes at the same time
- A class can inherit from a subclass of another class
- A class can be subclass of one and superclass of another simultaneously.

9.1.3

Method Resolution Order

Multiple inheritance is a powerful mechanism, but little is used, and in fact many object-oriented languages, such as Java, do not offer it because it is very complex to implement.

When the message method of the Lower class calls the message method of its superclass What method should be called, that of Intermediate1 or that of Intermediate2?

The solution varies by language. Python applies an algorithm to calculate the order of resolution of methods (methods resolution order, mro) whose result can be known using the class method .mro().

```
order = Lower.mro()
```

result:

```
[
    <class '__main__.Lower'>,
    <class '__main__.Intermediate1'>,
    <class '__main__.Intermediate2'>,
    <class '__main__.Upper'>, <class 'object'>
]
```

9.1.4

Complete the following instruction by calling the appropriate method to display the order of resolution of methods of the Lower class:

```
order = Lower._____
```

9.1.5

mro algorithm

To illustrate the algorithm that calculates the methods resolution order (mro) in Python, we will use the following class hierarchy.

The resolution list, L, of class E (C, D) is calculated as:

```
L[E(C, D)] = E + merge(L[C(A, B)], L[D(A, B)], [C, D])
```

The mixture is solved with the following algorithm:

1. The first element of the first list is taken; in example L [C] [0]
2. If this element is not in the "queue" of any list, add it to the result and delete it from the lists to be mixed; if not, try the first item in the following list ("queue" means the sublist consisting of all the elements except the first)
3. Repeat the previous operations until all the elements have passed to the result or no valid element is found according to the conditions of the previous point

In our example:

```
L[E(C, D)] = E + merge(L[C(A, B)], L[D(A, B)], [C, D]) =
    E + merge(C + merge(L[A], L[B], [A, B]), D + merge(L[A],
L[B], [A, B]), [C, D]) =
    E + merge([C, A, B], [D, A, B], [C, D]) =
    E + [C + merge([A, B], [D, A, B], [D])] =
    E + [C + [D + merge([A, B], [A, B])]] =
    E + [C + [D + [A + merge([B], [B])]]] =
    E + [C + [D + [A + [B]]]] = [E, C, D, A, B]
```

Note that if the order of inheritance in class D had been established as D (B, A), it could not have been resolved:

```
L[E(C, D)] = E + merge(L[C(A, B)], L[D(B, A)], [C, D]) =
    E + merge(C + merge(L[A], L[B], [A, B]), D + merge(L[B],
L[A], [A, B]), [C, D]) =
    E + merge([C, A, B], [D, B, A], [C, D]) =
    E + [C + merge([A, B], [D, B, A], [D])] =
    E + [C + [D + merge([A, B], [B, A])]] =
    There is no item that is first in a list and not in the
queue of another!
```

IMPORTANT: Although not included to simplify the explanation, all lists should end with the *object* class, which in Python 3 is the root of any class hierarchy.

9.1.6

Given the following statements:

```
class A: pass
class B: pass
class C(A,B): pass
class D(A,B): pass
class E(C, A): pass
```

What is the mro list of class E?

- [E, C, A, B, Object]
- [E, C, D, A, B, Object]
- The mro of class E can't be resolved

📖 9.1.7

Abstract classes

An abstract class is a class that contains abstract methods, which are declared but not implemented methods. In Python, abstract classes must be derived from the *ABC* (Abstract Base Class) class and the abstract methods must be marked with the *@abstractmethod* decorator. In the following example the *area* property is declared using an abstract method. Note that the mere fact that the method contains only the pass instruction does not make it abstract:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def area(self): pass
```

You cannot create objects of an abstract class. Abstract classes only serve to define a set of methods as a "contract" to which other classes can adhere, how? declaring heirs of the abstract class, which forces them to implement all the abstract methods declared in it. The *Square* class in the following example is a concrete subclass of the abstract *Shape* class and can be instantiated:

```
class Square(Shape):
    def __init__(self, side):
        self.__side = side

    @property
    def area(self):
        return self.__side ** 2
```

A subclass of an abstract class that does not implement all the abstract methods declared in that class is still abstract and cannot be instantiated.

An abstract class can contain non-abstract methods, as well as other data attributes.

9.1.8

What is the name of the class from which any abstract class should derive in Python?

Iterators and Generators

Chapter 10

10.1 Iterators and generators

10.1.1

Iterators

In Python, an iterator is an object that returns data, one element at a time. To do this it must define two magic methods: *__iter__* and *__next__*. The following example implements a class that allows iterating in a sequence of numbers within a range and separated by a predefined step.

```
class MyRange():
    def __init__(self, start=0, stop=0, step = 1):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        self.current = self.start
        return self

    def __next__(self):
        if self.current <= self.stop:
            result = self.current
            self.current += self.step
            return result
        else:
            raise StopIteration
```

The *__init__* method simply sets the parameters for iteration, the *__iter__* method prepares and returns the object to start the iteration, and the *__next__* method returns the current element and moves to the next one. The *StopIteration* exception marks the end of the iteration. We can make an infinite iterator if we do not raise that exception, but we must manage it carefully

A major benefit of iterators is that we can to treat huge datasets, one element at a time, without waste a big amount of memory storing the whole dataset.

10.1.2

Which is true?

- In Python, an iterator is an object that returns data, one element at a time.
- In Python, an iterator is a sort of loop defined using special magic methods.

- In Python, an iterator is an object that loops upon a range of numbers, treating one at a time.

10.1.3

The built-in functions *iter* and *next*

We can use an iterator wherever an iterable object is required, like in a for loop:

```
for i in MyRange(5, 20, 3):
    print(i) # Prints the sequence 5, 8, 11, 14, 17, 20
For-loops manage iterators by means of two built-in
functions, iter and next, which relies on the corresponding
magic methods __iter__ and __next__. A loop like the
following:for i in MyRange(5, 20, 3):
    print(i)
```

Really works like:

```
iter_obj = iter(MyRange(5, 20 ,3)) # iterator is created

while True:
    try:
        i = next(iter_obj) # gets next value
        print(i)
    except StopIteration:
        break  # if StopIteration is raised, break from loop
```

As the above example shows, we can iterate at the pace we want by calling the iter and next functions directly when necessary.

10.1.4

Which methods must be defined to convert a class in an iterator?

- __iter__
- __next__
- iter
- next

10.1.5

Generators

In Python, generators are an easy way to define iterators. They look like normal functions that replace return statements by *yield* statements.

```
def range_generator(start, stop, step = 1):
    current = start

    while current <= stop:
        yield current
        current += step
```

The first execution of a generator returns an iterator that then can be iterate using the *next* function. Each call to next executes the generator function until the next *yield* statement. A *yield* statement pauses the execution of the function and returns a value. The execution can be resumed later using the *next* function.

```
iter_obj = range_generator(3, 20 ,3)

while True:
    try:
        i = next(iter_obj)
        print(i)
    except StopIteration:
        break
```

Generators can be used wherever an iterator is required, including for-loops.

```
for i in range_generator(3, 20, 3):
    print(i)
```

10.1.6

Which statement is required to define a generator?

- yield
- iter
- next

10.2 Iterators (exercises)

10.2.1 Evens iterator

Write an iterator named *Evens* which returns, one at a time and in ascending order, all the even numbers greater or equal than 2 and less or equal than a number passed as a parameter to the initializer.

Example of use:

```
for i in evens(10):
   print(i)

Output:
2
4
6
8
10
--------------------
for i in evens(15):
   print(i)

Output:
2
4
6
8
10
12
14
```

evens.py

```
class Evens:
    """iterator: returns even numbers from 2 upto a given 
limit"""

    def __init__(self, stop):
        """Initialize iteration stop limit"""
        # write your code

    def __iter__(self):
        """Prepare iteration"""
        # write your code
    def __next__(self):
        """Advance until reach the stop limit"""
        # write your code

if __name__ == "__main__":
    for i in Evens(10):
        print(i)
```

⌨ 10.2.2 Powers of two iterator

Write an iterator named *PowersOfTwo* which returns, one at a time and in ascending order, the first n powers of two.

Example of use:

```
for i in PowersOfTwo(5):
   print(i)

Output:
0
2
4
16
32
------------------------
for i in PowersOfTwo(7):
   print(i)

Output:
1
2
4
16
32
64
128
```

⌨ 10.2.3 Divisors iterator

Complete the *Divisors* class to turn it into an iterator that returns, one by one and in ascending order, all the divisors of a positive integer.

Example of use:

```
for i in Divisors(5):
   print(i)

Output:
1
5
----------------------
for i in Divisors(12):
   print(i)

Output:
1
2
3
4
6
12
```

divisors.py

```
class Divisors:
    """iterator: Returns, one at a time, the Divisors os a
positive integer"""

    def __init__(self, number):
        """Set number to get its Divisors
           raise ValueError if number is not a positve integer
        """
        if type(number) == int and number > 0:
            self.__number = number
        else:
            raise ValueError("Initalization parameter must be
a positive integer")

    # Put your code here

if __name__ == "__main__":
    for i in Divisors(10):
        print(i)
```

10.2.4 Fibonacci iterator

The Fibonacci sequence is a sequence of numbers such that each number is the sum of the two preceding ones, starting from 0 and 1. That is:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

Complete the *Fibonacci* class to be an iterator that returns, one by one and in ascending order, the first *n* numbers of the Fibonacci sequence, being *n* a positive number.

Example of use:

```
for i in Fibonacci(5):
   print(i)

Output:
0
1
1
2
3
-----------------------
for i in Fibonacci(10):
   print(i)

Output:
0
1
1
2
3
5
8
13
21
34
```

fibonacci.py

```
class Fibonacci:
    """iterator: Returns, one at a time, the first n Fibonacci
numbers"""

    def __init__(self, n):
        """Set the stop condition
        """
        if type(n) == int and n > 0:
            self.__n = n
        else:
            raise ValueError("Initalization parameter must be
a positive integer")

    # Put your code here

if __name__ == "__main__":
    for i in Fibonacci(10):
        print(i)
```

10.3 Generators (exercises)

10.3.1 Divisors generator

Write a generator named *Divisors* that returns, one by one and in ascending order, all the divisors of a positive integer.

Example of use:

```
for i in Divisors(5):
   print(i)

Output:
1
5
----------------------
for i in Divisors(12):
   print(i)
Output:
1
2
3
4
6
12
```

10.3.2 Fibonacci generator

The Fibonacci sequence is a sequence of numbers such that each number is the sum of the two preceding ones, starting from 0 and 1. That is:

```
F(0) = 0
F(1) = 1
```

```
F(n) = F(n-1) + F(n-2)
```

Define a *Fibonacci* generator that returns, one by one and in ascending order, the first *n* numbers of the Fibonacci sequence, being *n* a positive number.

Example of use:

```
for i in Fibonacci(5):
   print(i)

Output:
0
1
1
2
3
-----------------------
for i in Fibonacci(10):
   print(i)
Output:
0
1
1
2
3
5
8
13
21
34
```

Custom Exceptions, Assertions

Chapter 11

11.1 Custom exceptions, assertions

11.1.1

How to define new exceptions

When an abnormal situation occurs during the execution of a program (for example, an attempt to divide by zero or access a file that does not exist) an exception is raised. In this context, an exception is a special object that carries information about the error.

Python has many classes of built-in exceptions, however, there are situations in which it is necessary to define a new exception class to precisely indicate a type of error that may arise when executing a program in the context of a specific problem. An exception class must derive, directly or indirectly, from the built-in class *Exception*:

```
class MyOwnExceptionError(Exception):
    pass
```

Usually, the classes that represent exceptions are given names that end with the word "Error", by similarity with the names of the built-in exceptions. Once defined, a custom exception can be thrown as it is done with a built-in exception:

```
if condition:
    raise MyOwnExceptionError()
```

An explanatory error message can be included:

```
if condition:
    raise MyOwnExceptionError("Error message")
```

11.1.2

Which is true?

- Custom exception classes must be derived from the built-in Exception class
- Custom exception classes must be given a name ending with the word "Error"
- Custom exception classes must include an explanatory message when created

📖 11.1.3

Customizing custom exceptions

An exception class is like any other class and can include data attributes and methods, usually for the purpose of providing additional information about the exception.

```
class MyOwnExceptionError(Exception):
    def __init__(self, obj, message):
        super().__init__(message)
        self.obj = obj
```

Although this may be useful at times, the best action is to keep the definition of the exception as simple as possible.

📝 11.1.4

Exception classes can be defined which do anything any other class can do

- True
- False

📖 11.1.5

Hierarchies of exceptions

When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions, so creating a hierarchy:

```
class DateError(Exception):
    """Base class for erroneus dates"""
    pass

class MonthError(DateError):
    """Raised when you try to create a date with a month value
       that is not between 1 and 12
    """
    pass
```

```
class DayError(DateError):
    """Raised when you try to create a date with a day value
which does not match
       with the month value
    """
```

In the above example, we have created a custom exception called DateError which is derived from the Exception class and serves as the base class for our own hierarchy, composed of the MonthError and DayError exceptions. This hierarchy is supposed to be used in a module offering a class to represent dates (mm/dd/yyyy).

11.1.6

When creating a module that can raise several distinct errors...

- a common practice is to create a hierarchy for exceptions defined by that module
- a common practice is to create a single class for exceptions defined by that module
- a common practice is to create a hierarchy for classes defined by that module

11.1.7

Assertions

Assertions are statements that establish a condition that must be true in a point of a program to continue execution. for example, if we are going to divide two numbers it must be true that the divisor is not equal to zero.

When the program execution reaches an assertion, the condition is evaluated. If it is true, the execution continues, else the program stops.

Python has a built-in *assert* statement to write assertions in a program:

```
def average(items):
    assert len(items) != 0, "Cannot calculate the average of
an empty list"
    return sum(items) / len(items)
```

The above example shows a function that calculates the average of the elements of a list, for which the list must have at least one element.

The assert statement that is just before the calculation of the average ensures that this condition is met, or the program is stopped by raising an *AssertionError* exception.

The assert declaration message is optional. The assert statement is executed only if the built-in variable *__debug__* is true, which is the case unless the Python interpreter has been launched with the optimization option *-o*.

The previous example is equivalent to:

```
def average(items):
    if __debug__:
        if len(items) == 0:
            raise AssertionError("Cannot calculate the average
of an empty list")
    return sum(items) / len(items)
```

11.1.8

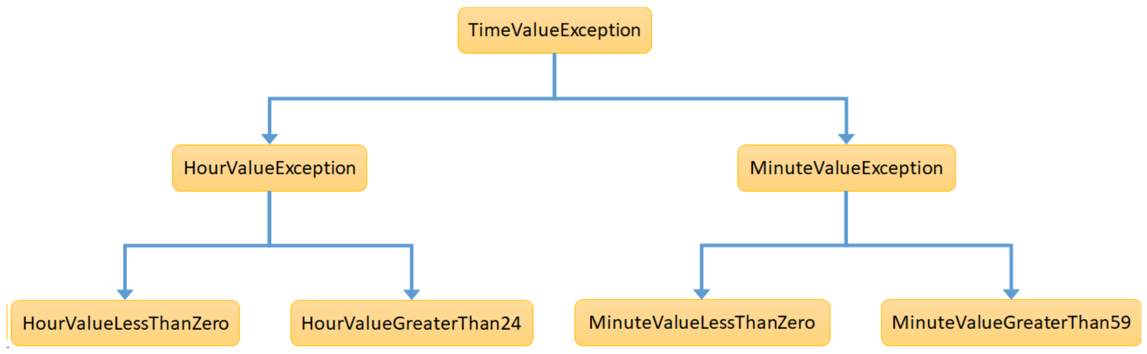
Which is true?

- An assert statement raises an exception when its condition is false
- An assert statement stops the execution of a program when its condition is true
- An assert statement stops the execution of a program when an exception is raised

11.2 Custom exceptions, assertions (exercises)

11.2.1 Time exceptions

Define, in the file time_exceptions.py, the following hierarchy of exceptions:



mytime.py

```
import time_exceptions

class Time:
    """Represents a time"""
    def __init__(self, h, m):
        """Time objects are initialized with hours and
minutes"""
        self.hour = h
        self.minute = m

    @property
    def hour(self):
        return self.hour

    @hour.setter
    def hour(self, value):
        if (type(value) == int and value >= 0 and value <|=
23):
            self.__hour = value
        elif type(value) != int:
            raise time_exceptions.HourException
        elif value <| 0:
            raise time_exceptions.HourValueLessThanZero
        elif value > 59:
            raise time_exceptions.HourValueGreaterThan24
        else:
            raise time_exceptions.HourValueException


    @property
    def minute(self):
        return self.minute

    @hour.setter
    def minute(self, value):
        if (type(value) == int and value >= 0 and value <|=
59):
            self.__minute = value
            self.__hour = value
        elif type(value) != int:
            raise time_exceptions.MinuteException
        elif value <| 0:
            raise time_exceptions.MinuteValueLessThanZero
        elif value > 59:
            raise time_exceptions.MinuteValueGreaterThan59
        else:
            raise time_exceptions.MinuteValueException

if __name__ == "__main__":
    # Example of use

    try:
        t = Time(102, 5)
    except time_exceptions.TimeValueException as e:
        print(type(e).__name__, " raised")
```

11.2.2 Time assertions

The *mytime* module contains the *Time* class, a class to represent hours and minutes. The initialization of an object of the *Time* class requires two parameters: one for the hours and another for the minutes, in this order. The hours are assumed to be an integer value between 0 and 23 and the minutes an integer value between 0 and 59.

Modify the setters of the *hour* and *minute* properties to assure, using assertions, that *hour* is an integer value between 0 and 23 and *minute* is an integer value between 0 and 59. In case any of these conditions fails, an *AssertionError* must be raised.

mytime.py

```
class Time:
    """A time with hours and minutes"""
    def __init__(self, h, m):
        self.hour = h
        self.minute = m

    @property
    def hour(self):
        return self.__hour

    @hour.setter
    def hour(self, h):
        self.__hour = h

    @property
    def minute(self):
        return self.__minute
    @minute.setter
    def minute(self, m):
        self.__minute = m

    def __str__(self):
        return str(self.hour).zfill(2) + ":" +
str(self.minute).zfill(2)

if __name__ == "__main__":
    h = Time(24, 12)
```

11.2.3 Fibonacci generator assertions

The Fibonacci sequence is a sequence of numbers such that each number is the sum of the two preceding ones, starting from 0 and 1. That is:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

The *fibonacci* generator returns, one by one and in ascending order, the first *n* numbers of the Fibonacci sequence, being *n* a positive number.

Add assertions to the *fibonacci* generator to ensure that the input parameter *n* is of type int and has a value equal or greater than 0.

fibonacci.py

```
def fibonacci(n):
    """Returns, one at a time, the first n Fibonacci
numbers"""

    fibo_1 = 0
    yield fibo_1

    if n > 1:
        fibo_2 = 1
        yield fibo_2

        current = 2

        while current <| n:
            current += 1
            current_fibo = fibo_1 + fibo_2
            fibo_1 = fibo_2
            fibo_2 = current_fibo
            yield current_fibo

if __name__ == "__main__":
    # Example of use (Not part of the solution)
    for i in fibonacci(10):
        print(i)
```

PRISCILLA

priscilla.fitped.eu