Chapter 8: Object-oriented programming

Python is an *object-oriented* programming language (OOP). **Objects** are an encapsulation of variables and functions into a single entity.

Let's assume we have a data type for rectangles. Without objects we could write the following code:

In [1]:

```
rec1_bl = (0, 2) # bl = bottom-left
rec1_ur = (6, 8) # ur = upper-right

rec2_bl = (4, 3)
rec2_ur = (7, 5)

def area(bl, ur):
    width = ur[0] - bl[0]
    height = ur[1] - bl[1]
    return width * height

print("Area of rectangle #1: {0}".format(area(rec1_bl, rec1_ur)))
print("Area of rectangle #2: {0}".format(area(rec2_bl, rec2_ur)))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
```

As we can observe the data (`rec1_bl` , `rec1_ur` , etc.) and the functions (`area()` and possible further functions) are defined separately, not encapsulated together.

Classes and objects

Simply put, an object is a collection of data (variables) and methods (functions) that act on those data. A **class** is a blueprint for the object. Classes introduce new data types in Python, describing real-world things and situations. Objects are the *instances* of classes, the creation process of objects are also called *instantiation*.

Let's create the `Rectangle` class now:

In [2]:

```
class Rectangle():
    name = 'Rectangle'

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle()
rec1.bl = (0, 2)
rec1.ur = (6, 8)
rec2 = Rectangle()
rec2.bl = (4, 3)
rec2.ur = (7, 5)

print("Area of rectangle {0}".format(rec1.area()))
print("Area of rectangle {0}".format(rec2.area()))
```

```
Area of rectangle 36
Area of rectangle 6
```

In this example the `Rectangle` class has 4 attributes: `name`, `ur`, `bl` and `area()`. Attributes may be data or method: the `name` is a simple string, `bl` and `ur` are tuples while `area()` is a function. Functions in a class are called *methods* more specifically.

The `rec1 = Rectangle()` statement creates a new instance object named `rec1` from the class `Rectangle`. We can access the attributes of objects using the object name prefix, e.g. `rec1.area()`.

Remember how we used list and dictionary functions:

```
numbers = [1, 4, 5, -2, 8]
numbers.sort()

shopping_list = {'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
for item in shopping_list.items():
    print(item)
```

This is the same syntax, we are calling methods on objects.

`self` parameter

There is a `self` parameter in the `area()` function definition inside the `Rectangle` class but, we called the method simply as `rec1.area()`, without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `rec1.area()` translates into `Rectangle.area(rec1)`.

In [3]:

```
print("Area of rectangle #1: {0}".format(Rectangle.area(rec1)))
print("Area of rectangle #2: {0}".format(Rectangle.area(rec2)))
print("Name of all rectangles: {0}".format(Rectangle.name))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
Name of all rectangles: Rectangle
```

In general, calling a method with a list of arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called `self`. It can be named otherwise but it is highly discouraged to follow the convention.

Constructors

In our previous example we deliberately gave a value to the `bl` and `ur` attributes of `rec1` and `rec2` before calling the `area()` method on them, so it can process those values. What happens if we e.g. forget to initialize those attributes beforehand?

In [4]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-4-a77eecff7f61> in <module>
      1 rec3 = Rectangle()
----> 2 print("Area of rectangle #3: {0}".format(rec3.area()))

<ipython-input-2-1afbe4a6155f> in area(self)
      3
      4     def area(self):
----> 5         width = self.ur[0] - self.bl[0]
      6         height = self.ur[1] - self.bl[1]
      7         return width * height

AttributeError: 'Rectangle' object has no attribute 'ur'
```

This issue can be addressed with a special *constructor method*, which is always executed when a new object is instantiated from a class.

In Python, class functions that begins with double underscore (`__`) are called special functions as they have special meaning. The `__init__()` function has particular interest for us now. This special function gets called whenever a new object of that class is instantiated. This type of function is also called a **constructor** in object-oriented programming. We normally use it to initialize all the variables.

In [5]:

```
class Rectangle():
    name = 'Rectangle'

    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle(0, 2, 6, 8)
rec2 = Rectangle(4, 3, 7, 5)

print("Area of rectangle #1: {0}".format(rec1.area()))
print("Area of rectangle #2: {0}".format(rec2.area()))
```

```
Area of rectangle #1: 36
Area of rectangle #2: 6
```

Now we cannot "forget" to pass all the required data to the object upon instatiation, because Python will raise a `TypeError` .

In [6]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-6-a77eecff7f61> in <module>
----> 1 rec3 = Rectangle()
      2 print("Area of rectangle #3: {0}".format(rec3.area()))

TypeError: __init__() missing 4 required positional arguments: 'bl_
x', 'bl_y', 'ur_x', and 'ur_y'
```

Default arguments

Alternatively we could use default values for the parameters, so a `Rectangle` could be constructed without defining its dimensions, but still giving value to the *instance attributes*.

In [7]:

```
class Rectangle():
    name = 'Rectangle'

    def __init__(self, bl_x = 0, bl_y = 0, ur_x = 0, ur_y = 0):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height
```

In [8]:

```
rec3 = Rectangle()
print("Area of rectangle #3: {0}".format(rec3.area()))
```

```
Area of rectangle #3: 0
```

Class and instance attributes

Generally speaking, instance attributes are for data unique to each instance and class attributes are for variables and methods shared by all instances of the class.

In the example `Rectangle` class, the `name` attribute is a class variable, because it is defined as an attribute of the class.

In [9]:

```
print(Rectangle.name)
```

```
Rectangle
```

The `bl` and `ur` are instance attributes (because they are accessed through the `self` object). This means that each rectangle can have its own bottom-left and upper-right position, but all rectangles share the same name.

In [10]:

```
rec1.bl = (-2, 1)
print(rec1.bl) # has no effect on rec2
print(rec2.bl)
```

```
(-2, 1)
(4, 3)
```

String representation of an object

By default, the string representation of an object consists of the type name and memory address:

In [11]:

```
print(rec1)
```

```
<__main__.Rectangle object at 0x7fbcfd0291f0>
```

As we discussed, methods that begins with double underscore (`__`) are called special functions in Python. The `__str__()` method is another special function, which can compute and return the "informal" or nicely printable string representation of an object. The return value must be a string object.

In [12]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

rec1 = Rectangle(0, 2, 6, 8)
rec2 = Rectangle(4, 3, 7, 5)

print(rec1)
print(rec2)
```

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

Summary exercises on object-oriented programming

Task 1: Perimeter

Extend the `Rectangle` class with a `perimeter()` method.

Sample usage:

```
result = rec1.perimeter()
# result is an integer value
```

In [13]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def area(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return width * height

    def perimeter(self):
        width = self.ur[0] - self.bl[0]
        height = self.ur[1] - self.bl[1]
        return 2 * (width + height)
```

The computation of the width and height of the rectangle is now redundantly given in the `area()` and the `perimeters()` methods. Eliminate the redundancy by extracting a new `width()` and `height()` function in the `Rectangle` class.

In [14]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())
```

Task 2: Translation

Extend the `Rectangle` class with a `translate()` method, which moves it in the Euclidean space in the given direction.

Sample usage:

```
print(rec1)
rec1.translate(3, 4)
print(rec1)
```

In [15]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())

    def translate(self, x, y):
        self.bl = (self.bl[0] + x, self.bl[1] + y)
        self.ur = (self.ur[0] + x, self.ur[1] + y)
```

Task 3: Overlap

Extend the `Rectangle` class with an `overlap()` method, which can decide whether 2 rectangles overlap each other.

Sample usage:

```
result = rec1.overlap(rec2)
# result is a boolean value
```

In [16]:

```
class Rectangle():
    def __init__(self, bl_x, bl_y, ur_x, ur_y):
        self.bl = (bl_x, bl_y)
        self.ur = (ur_x, ur_y)

    def __str__(self):
        return "Rectangle ({0}, {1}, {2}, {3})".format(self.bl[0], self.bl[1], s
elf.ur[0], self.ur[1])

    def width(self):
        return self.ur[0] - self.bl[0]

    def height(self):
        return self.ur[1] - self.bl[1]

    def area(self):
        return self.width() * self.height()

    def perimeter(self):
        return 2 * (self.width() + self.height())

    def translate(self, x, y):
        self.bl = (self.bl[0] + x, self.bl[1] + y)
        self.ur = (self.ur[0] + x, self.ur[1] + y)

    def overlap(self, other):
        overlap_x = ((self.bl[0] < other.bl[0] and self.ur[0] > other.bl[0]) or
                     (other.bl[0] < self.bl[0] and other.ur[0] > self.bl[0]))
        overlap_y = ((self.bl[1] < other.bl[1] and self.ur[1] > other.bl[1]) or
                     (other.bl[1] < self.bl[1] and other.ur[1] > self.bl[1]))

        return overlap_x and overlap_y
```

Hint: Two *axis-parallel* rectangles overlap if they overlap either by the X or Y dimensions.
They overlap by the X dimension if `A_X1 < B_X1` and `A_X2 > B_X1` ; or `B_X1 < A_X1` and `B_X2 > A_X1` .
Similar inequality condition apply on the Y dimension.

Data classes *(optional)*

A relatively new feature available since Python 3.7 (released in 2018) is the *data class*. A data class is a class typically containing mainly data, although there aren't really any restrictions. It is created using the `@dataclass` decorator, as follows:

In [17]:

```
from dataclasses import dataclass

@dataclass
class Country:
    name: str
    capital: str
    area: int
    population: int
    gdp: int
    literacy: float
    region: str = 'Unknown'

    def population_density(self):
        return self.population / self.area
```

The benefit of using data classes is that some special methods, e.g. the `__init__()` constructor method will be automatically generated and added to the class, initiating all instance variables. The generated constructor will look like:

```
    def __init__(self, name, capital, area, population, gdp, literacy, region
     = 'Unknown'):
        self.name = name
        self.capital = capital
        self.area = area
        self.population = population
        self.gdp =gdp
        self.literacy = literacy
        self.region = region
```

Since data classes is just a new syntactical approach in Python for defining classes, we can instantiate objects from data classes and use them just like before:

In [18]:

```
hungary = Country('Hungary', 'Budapest', 93030, 9981334, 13900, 99.4, 'Central-E
urope')
```

In [19]:

```
print(hungary.capital)
print(hungary.population_density())
```

```
Budapest
107.29156186176502
```

The string representational `__str__()` method is also predefined for data classes:

In [20]:

```
print(hungary)
```

```
Country(name='Hungary', capital='Budapest', area=93030, population=9
981334, gdp=13900, literacy=99.4, region='Central-Europe')
```

Type annotations

As you have may noticed we also defined the type of the instance variables in the data class, e.g. `population: int` . This is called *type hinting* or *type annotations* and is mandatory when defining a data class. Type hinting is available in Python since version 3.5 and can also be used elsewhere: for local variables, function parameters, return types, etc.

Note that the Python runtime does not enforce function and variable type annotations, so whether you use them or not, they will not affect how your code is executed. However they can be used by third party tools such as type checkers (see mypy (http://mypy-lang.org/)) or integrated development environments (IDEs), to early detect potential errors in your code.

We will not use type hinting further in this course, as *Jupyter Notebook* itself does not perform type checking based on them.

Inheritance *(advanced, optional)*

We do not always have to start from scratch when writing a class. If the class is a specialized version of another already existing class, we can use **inheritance**.

When one class *inherits* from another, it automatically takes on all the attributes and methods of the first class. The original class is called the *parent* class, and the new class is the *child* class. The child class inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

Let's inherit the `Square` class from the `Rectangle` class:

In [21]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
```

```
Area of square #1: 9
```

The `__init__()` function in the child class

Often we would like to reuse the original `__init__` function of the parent class in the child class.

This can be done with the `super()` function inside the child class constructor. This is a special function that helps Python make connections between the parent and child class.

Note: The name super comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

In [22]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        super().__init__(bl_x, bl_y, bl_x + width, bl_y + width)

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
```

```
Area of square #1: 9
```

Overiding methods from the *parent* class

Let's see what happens if we print our `s1` object:

In [23]:

```
print(s1)
```

```
Rectangle (5, 10, 8, 13)
```

It shows the text "Rectangle", because the `__str__()` special function was defined this way in the `Rectangle` parent class and now the `Square` child class inherited it.

We can override any method from the parent class that do not fit into the model of the child class. To achieve this, we can simply redefine the method in the child class with the same name as the method we want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

In [24]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

    def __str__(self):
        return "Square ({0}, {1}, width = {2})".format(self.bl[0], self.bl[1], s
elf.ur[0] - self.bl[0])

s1 = Square(5, 10, 3)
print("Area of square #1: {0}".format(s1.area()))
print(s1)
```

```
Area of square #1: 9
Square (5, 10, width = 3)
```

Note: the `super()` function can be used in any overriding child class methods.

Extend the functionality of the parent class

Child classes may also extend the functionality of their parent class by adding new methods to themselves.

In [25]:

```
class Square(Rectangle):
    def __init__(self, bl_x, bl_y, width):
        self.bl = (bl_x, bl_y)
        self.ur = (bl_x + width, bl_y + width)

    def __str__(self):
        return "Square ({0}, {1}, width = {2})".format(self.bl[0], self.bl[1], s
elf.ur[0] - self.bl[0])

    def side(self):
        return self.bl[1] - self.ur[0]
```

In [26]:

```
s1 = Square(5, 10, 3)
print(s1.side())
```

```
2
```

The `Rectangle` class does not have this new `side()` method:

In [27]:

```
print(rec1.side())
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-27-63985b0fc09e> in <module>
----> 1 print(rec1.side())

AttributeError: 'Rectangle' object has no attribute 'side'
```