Object-Oriented Programming: A Very Short Introduction

What is Object Oriented Programming?

TLdr; OOP is a programming paradigm where objects rather than functions are the main entities of a script.

Functions vs. Objects

Some of you might have already encountered objects while writing Python code. However, for those that did not yet encountered objects, let us briefly discuss how objects differ from a well-known entity: ***functions***.

As you know functions in Python usually look like this[1]:

```
def add_ten(x):
      added = x + 10
      return added
```

Thus, we have a `def` followed by the name of the function and its arguments. Then we have the body of the function - in this case `added = x + 10`. At the end, there is a `return` statement. This function works well when we have two values that we want to add. However, what would happen for situations in which we have a list of 20 values for which we want to 10 to each value in the list. Potentially, we might end up[2] with a script that looks something like this:

```
val_list = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

for value in val_list:
      add_ten(value)
```

Obviously, this loop calls the function `add_ten()` 20 times. As a result, the function will

[1] The code theme used here is atelier-heath-light.
[2] For educational purposes, we will avoid using the append-method for now.

be called 20 times and it will, as a result, output a value 20 times. However, this is not very efficient. Not only is it expensive to call a function 20 times, one can imagine that as the size of the input data increases, the time needed to compute this, increases significantly. There are, of course, more efficient ways to solve this problem: building objects!

Building Objects I

Having discussed why objects can be useful, we can now proceed with actually building a Python object. The first thing we do is to define a `class`. A class is an object in Python that can be used to create your own Python objects. First of all, you should give the class a name. Similar to other entities in Python we use a colon and proceed to the next line. Important to note here is that the class - contrary to functions - do not take any input arguments. However, this does not mean that nothing is fixed beforehand. The next step, namely, is to initialise the variables. We do this using the `def __init__(self)`. This part of the class sometimes called a ***constructor*** and needs to be declared explicitly at the beginning of each class. This might look like as follows:

```
class Horse:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def neigh(self):
        print(self.name, "the horse neighs.")

    def be_awesome(self):
        print(self.name, "the horse is being awesome.")

    def age_horse(self):
        print(self.name, "is", self.age, "years old.")

    def year_born(self):
        print(self.name, "was born in the year", 2019-self.age)

def main():
    #Instantiate
    john = Horse('John',23)
```

```
    john.be_awesome()
    john.age_horse()
    john.year_born()

if __name__ == "__main__":
    main()
```

This class is called `Horse`. The variables are initialised at the beginning of the class using the ***init-constructor***. Here, the class takes three variables: (1) `self`, (2) `name` and (3) `age`. The latter two variables are, in a way, optional. The `self` variable, on the other hand, is mandatory when you want to write a working Python object. The self variable is necessary because it lets you instantiate the object. By doing so you have it *refer to itself,* meaning that an instance of that class can be created. As a result, we initialize the variables `name` and `age` as `self.name = name` and `self.age = age`, thereby indicating that are, quite literally, dependent on the instance to which they belong. After all, my name and age will likely be different from yours. For instance john.name = john and john.age = 34, whereas for Anna we might have that anna.name = anna and anna.age = 53. Hence, the variables of a class can in some sense be understood as an indication of the characteristic or property of the object.

What this looks like in practice can be seen in the `main()` function where the object `john` is created using the `Horse` class. When we call the class Shark, we pass it a name (in this case 'John') as well as an age (in this case 23). Thus, John is a horse and John is 23 years old. This is shown below:

```
>> John the horse is being awesome.
>> John is 23 years old.
>> John was born in the year 1996
```

Once we have instantiated a class, we have an object of that class. Moreover, we can now call methods inside of that class. For instance, we can call the `year_born` method using `john.year_born()`. When calling this method, the method prints the John the Horse's birth year.

Note, moveover, that in this particular script, the instance calls are called from within a function called `main()`. This is a special function in Python that essentially lets you call all important functions and classes in one, clearly-demarcated block of code. Furthermore, there is the `if __name__ == "__main__:"` part of the code. This command goes hand in hand with the `main()` function. This part of the code is a special way of telling the compiler that the blocks of code that are not part of the

`main()` function should be executed. The if-statement is usually only used when a developer wants to execute a Python script and it is good practice to use it.

Building Objects II

The Horse object discussed above was a relatively easy example. Yet, in order to fully grasp why classes are useful, let us discuss another example. In this case, we will look at a class that contains a number of object operations such as summation, multiplication and division.

As before, we give the class a name and call it `Operations`. It is common (and good practice) to give classes names that start with a capital letter. Having named the class, we will again build a constructor to which we pass `self`, `x` and `y`, where `x` and `y` are numerical values (i.e. integers or floats).

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

    def summation(self):
        summa = self.x + self.y
        print("The sum of", self.x, "and", self.y, "is:", summa)

    def multiplication(self):
        multi = self.x * self.y
        print(self.x, "multiplied by", self.y, "is:", multi)

    def division(self):
        div = self.x - self.y
        print(self.x, "minus", self.y, "is:", div)

def main():
    numbers = Operations(1,2)
    numbers.summation()
    numbers.multiplication()
    numbers.division()

    numbers = Operations(6,31)
```

```
    numbers.summation()
    numbers.multiplication()
    numbers.division()

if __name__ == "__main__":
    main()
```

In this particular case, we define 3 methods: `summation`, `multiplication` and `division`. To reiterate, these are the methods we can call for some instance of the class. If we instantiate the class using the integers 23 for `x` and 7 for `y`, then calling `summation` will give us 30, whereas calling the `division` method will give us 16. What this looks like in practice is shown in the code snippet above, where for the first instance in the `main()` function the class is instantiated with 1 and 2. In the second instantiation 6 and 31 are used. The output of the code above, is shown below:

```
>> The sum of 1 and 2 is: 3
>> 1 multiplied by 2 is: 2
>> 1 minus 2 is: -1
>> The sum of 6 and 31 is: 37
>> 6 multiplied by 31 is: 186
>> 6 minus 31 is: -25
```

Building Objects III

Here we will discuss our final example. So far have only discussed classes for which the input consisted of singular items such as integers and strings. However, classes even more useful when there are other, much larger data types involved. Examples of such data-objects are data frames, large dictionaries or very large lists.

Below is an example of a class that has only one method: `new_append(self)`. This method returns an appended list.[3]

```
class NewAppend:

    def __init__(self, list_1, value):
        self.list_1 = list_1
        self.value = value
```

[3] N.B. As can be seen, a different list is returned from the one that is instantiated.

```
    def new_append(self):
        length = len(self.list_1)
        new_list = list(range(0, len(self.list_1) + 1))
        for i in range(len(new_list)):
            if i < len(self.list_1):
                new_list[i] = self.list_1[i]
            else:
                new_list[i] = self.value
        return new_list

def main():

    current_list = NewAppend([6,1,6,8,2],4)
    appended_list = current_list.new_append()
    print(appended_list)

    current_list = NewAppend([42,1,41,6,7,8,52,72,1],91)
    appended_list = current_list.new_append()
    print(appended_list)

if __name__ == "__main__":
    main()
```

As usual, we make a class. This time we call it `NewAppend`. Our class only has one method: `new_append`. The class can be instantiated by calling it and passing it a list as well as a value you want to append. By calling the `new_append` method, you can append the number to your input list. Thus, these specific instances output:

```
>> [6, 1, 6, 8, 2, 4]
>> [42, 1, 41, 6, 7, 8, 52, 72, 1, 91]
```