IMPORTANT NOTE: To properly use these notes you need to open them as a Jupyter Notebook. A .pdf form will not allow you to run some of the exercises contained inside the notes!

Classes: defining new Python objects from scratch

In this notebook, we will learn how to define new Python "objects", or so-called "classes".

A Python object is nothing but a **new data type, that can be generated from those which you are already familiar with** (lists, dictionaries, tuples, sets, ...) or, more generally, any previously defined object. For example, you can first define a class, then use it inside some other class's definition.

The possibility to define arbitrary objects/data types is what makes Python and other object-oriented languages extremely powerful and flexible!

Minimal declaration, the `__init__` function and data attributes

The **minimal** definition of a class should have the following form (with some caveats, see later in the final remarks):

```
class myNewClass( ):

   def __init__( self, input1, input2, ..., keywordArg1 = X, keywordArg2 = Y
, ...):
      self.dataAttributeX = input1
      self.dataAttributeY = input2
      ...
      self.aDefaultAttribute = value1
      self.anotherDefaultAttribute = value2
      return()
```

where `self` is literally the word `self` , whereas dataAttributeX, dataAttributeY, ... are so called *data attributes* of the class and `myNewClass` is the name of the new object we just defined. Note that attributeX, attributeY, ..., myNewClass could be any valid name of your choice. For example, I could have written `self.var1` instead of `self.dataAttributeX` and I could have called the class `bestObjectEver` instead of `myNewClass` .

Another thing to notice is that because the input to the `__init__` function ("double underscore init double underscore") are used to define values that will differ between between different instances of the same class (different variables of the same type of class), a class can also be provided with some default value that is general to all of them (the `value1` and `value2` in the declaration above.

A practical and simple example:

```
class person():

    def __init__( self, input1, input2, name = "Stefano" ):
        self.myAge = input1
        self.myGender = input2
        self.name = name
        self.money = 100

        return
```

Important note: the `__init__` function of a class should not provide any return value, so the last line is literally **only** `return` , and **nothing** else.

Once a class has been defined, you can create a variable with that data type. In Python jargon, the associated variable is called an
instance of the class (and similarly you can say the class has been "instantiated" once you assigned it to a specific variable). In other words, we created a Python object with the properties of the class.

The general declaration for instatiation goes like this:

```
varName = nameOfClass( input1, input2 )
```

so in our case, referring to the previous example:

```
person1 = person( 33, 'female', name = "Sandrine"  )
```

What we are doing in this way is to declare that `person1` is an instance of the object of type "person". Also, by this declaration we have assigned the value 33 to `input1` , the value "female" to input2 and we have over-written the default value of `name` with "Sandrine".

Once an object has been instantiated, we can retrieve the values of any data attribute with the general declaration:

```
nameOfclassInstance.attributeName
```

or, taking again the example from above (and assuming with have created the variable `person1` of type `person` as previously defined), `print( person1.name )` would print out "Sandrine", or `person1.age + 2` would give as a result `35` .

What about the value of `self` ???

In the previous example of class instantiation, you might have thought there was an error: the input values of the `__init__` function were `self, input1, input2, name` , but we have said the first value used in the class instantiation would be assigned to `input1` and the second to `input2` and you might wonder...what is the value of `self` ?

The fact is that, for the `__init__` function (and, actually *any* class-specific function, which we will see later) the first value in the function *declaration* is always the word `self` . However, when you make an *instance* of the class you **do not** need to provide its input value because `self` is automatically taken to be *the variable to which the class was instantiated*.

Confused? An example is better than a thousand words, so let's see one! First, read, and then run the cell below:

In [ ]:

```
class superClass():

    def __init__( self, input1, input2, input3 = "Hello" ):
        self.firstValue = input1
        self.second = input2
        self.some = input3

        return

var1 = superClass2( 1, 2 )
```

with the example above in mind, the value of `self` is now `var1`, whereas the values of the data attributes `firstValue` and `second` are 1 and 2, respectively. The data attribute `some` instead takes the default value of input3, which is thus `"Hello"`. Run the cell below to check this by yourself!

In [ ]:

```
print( var1.firstValue )
print( var1.second )
print( var1.some )

var2 = superClass( 44, 55, 66 )

print( var2.firstValue )
print( var2.second )
print( var2.some )
```

In practice, associating the name of the variable used for the class instantiation to the variable `self` is a useful way to be able to retrieve the data attributes of the class for that specific instantiation.

An example

Ok, let us now make a quick and simple exercise. In the next cell:

- Declare a new class whose name is "myFirstClass".
- This class must have 2 data attributes, one which we will call 'year' and an attribute which we will call 'length', with a default value of 5
- After the class declaration, make an instance of the class myFirstClass and call it 'thisClass'

If you do it correctly, when you run the next cells you should not get any error, and after the first cell, print out the value of the attribute 'length' (whatever value you have chosen when you declare the instance 'thisClass'), followed by 5. After running the second cell, you should see printed out the value 444, follow by the value of 'length'.

In [ ]:

```
class myFirstClass():

    def __init__( #REPLACE WITH YOUR CODE ):
        #REPLACE WITH YOUR CODE
        return

#REPLACE WITH YOUR CODE TO CREATE AN INSTANCE CALLED thisClass

print( thisClass.length )
print( thisClass.thisYear )

thisClass = myFirstClass( thisClass.length, 444 )
print( thisClass.thisYear )
print( thisClass.length )


```

One thing you should notice is that in giving you the previous exercise, I did not have to know the internal details of the class. In other words I did **not have to know exactly how you coded it up**! I just needed to know its data attributes, that's it.

The **idea of object-oriented programming** is somehow exactly what we just described: You should be **able to use an object just knowing the class attributes (assuming they have been coded correctly!), without needing to read the way it is constructed**.

This type of "coding phylosophy" is very powerful especially when different people collaborate on large projects. If I need to use some specific class coded by another person, the only thing we have to agree upon at the beginning is which attributes the class possesses, after that, I can start writing a piece of code using it even before the class has been defined!

Method attributes

Ok, so far, it was probably a bit...dull. I know. Let's see something more complex and introduce so called **method attributes** of a class.

Method attributes (or methods for short) are nothing but *functions that specifically apply to objects of the class*. If the variable `var1` is an instance of the class `nameOfTheClass` , i.e., if we have declared:

```
var1 = nameOfTheClass( list of required inputs )
```

and `nameOfTheClass` has some method attribute named `myMethod` , you can invoke it with the declaration

```
var1.myMethod( list of inputs for the method, if needed )
```

Notice this is basically the same way in which one can call a data attribute. However, since the latter is **not** a function, we do not need to call it with the parenthesis () - which indeed indicate a function - nor insert any input values inside them.

Actually, you should have noticed that *we have already seen one method attribute that **all** classes should have*: The function `__init__` (which is a short for "Initialize", as it defines the initial attributes of the class). In fact, the `__init__` method is special because unlike other methods of a class, it is always called as soon as you make an instance of that class. Other class methods instead must be explicitly invoked.

Let us extend the typical declaration of a class to see how it all works. This is the general construct:

```
class generalClass3():

    def __init__( self, input1, ... ):
        self.var1 = input1
        return

    def methodX( self, MethodInput1, ... ):  # Note that this is nothing but
a normal function!
        instruction1
        instruction2
        ...
        return( outputValue )
```

and let us also make immediately an explicit example in the cell below.

After reading the class declaration and the code below, **think about what you expect as outcome first**, then run the cell to see if this is indeed what you expected!

In [ ]:

```
class aComplexClass():

    def __init__( self, myList ):
        self.internal =  myList
        return

    def sortAndPrint( self ):
        '''Takes internal (a data attribute), supposed to be a list object,
        then sort it and print it'''
        self.internal.sort()
        for i in self.internal:
            print( '{0}'.format( i ) )
        print( 'End of List' )

        return

    def addAndPrint( self, myValue ):
        '''Takes internal (a data attribute), supposed to be a list object,
        and add "value" to all the elements'''
        for i in self.internal:
            i += myValue
            print( '{0}'.format( i ) )
        print( 'End of List' )
        return



aaa = aComplexClass( [ 3, 29, -1, 0, 24, 7 ] )

aaa.sortAndPrint()
aaa.addAndPrint( 3 )

```

A few important things to notice here:

- As also previously stated, when defining a method attribute, you **always** need to have the word `self` as the first argument(...but see the final remark at the end of these notes for a clarification). However, when *using* the method, the value of "self" is never read, because `self` is automatically given the name of the variable instantiated to the class.
- **Method attributes can use data attributes of the function previously assigned**. For example, the `sortAndPrint` method used the value of self.internal, which was previously assigned during instantiation
- **Method attributes can modify data attributes**. For example, because internal is a list, you can use the command "sort" on it. Sort also modifies the list itself permanently, once we later use the method "addAndPrint", this is being applied to the modified list, not the one given at instantiation.
- As any normal function, class methods *might* require additional inputs. For example, addAndPrint required you to also input the value of `myValue`.
- Generally speaking...we have seen classes before! In fact, we have used a few of the method attributes of objects of the class list, for example, the method ".sort()" was one. Similarly, all the functions / commands that we have been applying on lists, strings, dictionaries and so on are available because someone else has previously coded these classes for you!

A more complex exercise:

A somewhat complex exercise: Try to fill in the cell below to define an extra method, call it `sortAndElevate`, for the `aComplexClass` object. This method takes as input value `var1` and then:

1. sort the data attribute "internal" in ascending order
2. elevates all the elements of the ordered list to the power of var1 (without printing them)
3. return as output value the ordered and power-elevated list

However, after calling this method, "internal" must be exactly as the original, unsorted version initially provided during instantiation. If correct, when running the following cell it should print `List one: 1, 4, 9, 16` and then `Internal list: 2, 3, 1, 4`.

In [ ]:

```
class aComplexClass2():

    def __init__( self, myList ):
        self.internal =  myList
        return

    def sortAndPrint( self ):
        '''Takes internal (a data attribute), supposed to be a list object,
        then sort it and print it'''
        self.internal.sort()
        for i in self.internal:
            print( '{0}'.format( i ) )
        print( 'End of List' )

        return

    def addAndPrint( self, myValue ):
        '''Takes internal (a data attribute), supposed to be a list object,
        and add "value" to all the elements'''
        for i in self.internal:
            i += myValue
            print( '{0}'.format( i ) )
        print( 'End of List' )
        return

    def sortAndElevate( self, var1 ):
        # INSERT YOUR CODE HERE!
        return

bbb = aComplexClass2( [ 2, 3, 1, 4 ] )
aaa = aComplexClass.sortAndElevate( 2 )

print( 'List one', bbb.sortAndElevate( 2 ) )
print( 'Internal list:', bbb.internal )
```

The concept of "inheritance".

Inheritance in Python is a **very powerful tool to build new objects**, usually called *derived classes*, **based on properties of other objects** that have already been defined, without having to do all the work of re-writing the code. The general declaration is:

```
class nameOfDerivedClass( nameOfBasesClass ):
```

where `nameOfBasesClass` is any previously defined class. Let us immediately have a look at a simple example. Read the code and run the cell to see what happens (always think first, what do you expect?)

In [ ]:

```
class classOne():

    def __init__( self, myDictionary ):
        self.var1 = myDictionary
        return

    def printAllDef( self ):
        num = 1
        for i in self.var1.items():
            print( "The {0:d}th pair of this dictionary is {1}".format( num, i
            num += 1
        return

class classTwo( classOne ):

    def printReverse( self ):
        myCopy = self.var1.copy()
        for i in range( len( self.var1 ) ):
            b = self.var1.popitem()
            print( "This dictionary is reversed order is element:{0} = {1}".forr
        return

a = classOne( { 1:2, 2:4, 3:"a" } )
b = classTwo( { 1:2, 2:4, 3:"a" } )

b.printAllDef()
b.printReverse()
```

Note how we do not have to define any `__init__` method for a derived class, because it inherits it (as well as all other methods!) from the base class, also called parent class (and thus the derived class can be called its child).

The inherited methods can be used exactly in the same way they were used in the parent class. However, we can add new methods attributes, and extra data attributes too!

Re-defining methods in a derived class

At this point, you should ask yourself: what happens if we redefine a method (including `__init__`) ?

In practice, if you call a method for a specific object, the **first** thing the Python interpreter does is to look into the class definition. If it does not find anything, it looks into **any** class from which the object is derived, recursively (recursively because we can make a whole genealogic tree of classes, and make a derived class out of another derived class). **The first definition** of a method that is found in this search is the one used.

Another example

Let us make an example with some objects that you know quite well: lists!

Lists have a set of properties that you should remember. In the next example, we want to add some new ones in a new class that we will call "specialList", as well as re-defining the method "append" found in lists.

Read the class definition then run the cell below and check what happens.

In [ ]:

```
class specialList( list ):

    def intercalate( self, anotherList ):
        ''' Intercalates this list with the list anotherList.
        Example if you intercalate [1,2,3] and [4,5,6]
        you should expect to obtain [ 1, 4, 2, 5, 3, 6 ]. If the two lists have
        different lengths, return an error.
        '''
        if len( self ) != len( anotherList ):
            print( "Two lists can only be intercalated if of equal length!" )
            raise ValueError

        aaa = []
        for i in range( len( self ) ):
            aaa.append( self[ i ] )
            aaa.append( anotherList[ i ] )
        return(aaa)

    def append( self, x, number ):
        #Here we actually REDEFINE a method that already exists in
        #lists...check later what happens. Note that in normal lists
        #nameOfList.append( x ) takes x, whatever it is, and put it as the
        #last element of nameOfList...here we are doing something different,
        #check
        for i in range( number ):
            self += [ x ]
        return

special = specialList([ "a", "b", "c", "d" ])

second = [ 1, 2, 3, 4 ]

print( "Intercalated lists:", special.intercalate( second ) )

special.remove( "d" ) #Note we never defined this method attribute for specialL
                      #inherited from the list class from which it was derived!

special.append( "e", 3 ) #Note that this type of append is NOT the same as for
                      #we have redefined it for specialList

print( "New value of special", special )

print( special[::2]) #Even in this case, we are using a property of lists witho
                     #the class specialList!
```

A few extra remarks about classes

1. Be careful: **data attributes definitions override method attributes**. In other words, if a data attribute and a method attribute have the same name, the Python interpreter will always treat it following the definition of the data attribute. For this reason **make sure you never have a method and data attribute with the same name**! To clarify this point, see what happens when you run the following cell:

In [ ]:

```
# Read and run this cell

class dangerousNaming():

    def __init__( self, x ):
        self.myX = x
        return

    def myX( self ):
        print( self.myX**2 )
        return


var3 = dangerousNaming( 4 )

print( var3.myX() )
```

The previous program returns an error "int object is not callable"...have you understood why? The problem is that `myX` is a **data attribute** and **not** a method because. Because it is not a function, Python complains because you cannot call it with the parenthesis () at the end of its name! The mistake was that in the class declaration we had defined both a method and a data attribute with the name `myX` , but the data attribute takes precedence and over-rides the function definition.

If you did not really understand what is written above or are still confused, replace the command `print( var3.myX() )` with `print( var3.myX )` in the cell above and see what happens when you run it!

2. **Once you have instantiated a class object, you can add data attributes if you want**. However, these attributes will just belong to that specific instance, not to the general class itself. Read the cell below then run it to see an example (look at the error message to understand what is happening)!

In [ ]:

```

class anExample():

    def __init__( self, x ):
        self.myX = x
        return

var4 = anExample( 4 )

var4.mySecondX = 10

var5 = anExample( 4 )

print( var4.mySecondX )

print( var5.mySecondX )

```

3. If you have not noticed already before, **methods inside a class definition may call other methods**

defined for the same class by using method attributes of the "self" argument. Read and then run the next cell to see an example.

In [ ]:

```
class anotherExample():

    def __init__( self, x ):
        self.myX = x
        return

    def power2( self, input1 ):
        x = self.myX**2
        return x

    def power6( self, input1 ):
        j = self.power2( input1 )**3
        return j


var4 = anotherExample( x = 10 )

print( var4.power6( 10 ) )

```

4. A final remark...just to be precise. I have previously told you that the first argument of a method must be the word `self` ...but this is actually a (white, small) lie! In reality, the use of the word `self` is nothing more than a convention: the name `self` has absolutely no special meaning to Python (and in fact it is not a protected word).
   However, if you do not follow this convention, your code may be less readable to other Python programmers because **`self` is generally accepted by the Python community** as the first input to any class method. In this regard, you should notice that a "class browser program" that scans Python code looking for classes (for example, to optimise them), might be written relying upon such convention and thus fail to work with your code!