A BRIEF INTRODUCTION TO PYTHON

(With content based on the book “A Byte of Python”, by Swaroop [3])

A Little Object-Oriented Programming

- The programming style adopted so far has been essentially procedural or imperative
  - *but while this may be a common practice in implementations that we want simple and fast, in more complex problems it will make more sense to use object-oriented techniques – the ideal paradigm for dealing with complexity*
- As has been said before, Python supports the OOP paradigm, treating everything as objects
  - *same integers are seen as objects, instances of the int class – for more details, see help(int)*

In Python everything are Objects

- Class objects
  - *allow us to access all their members*

    Examples: MyClasse.__doc__, MyClass.x, x being an attribute (object instance) or a method (function object)
  - *and, of course, allow to instantiate "instance objects"*
    Example: obj=MyClass()
- Instance objects
  - *allow us to simply access their members*

    Example: obj.x and obj.f(), being x an attribute and f() a method
- Function/method objects
  - *a method, being an object, can be assigned to a variable, and invoked later referencing that variable*
  - *Example: y = obj.f()*

    print(y())

Encapsulation

- The Python creator decided not to include the protection mechanisms that are present in other languages, such as (C++, C# and Java), and that allow us to distinguish what is public from what is private
  - *Assuming that "we are all adults", leaves the programmer's responsibility to access only the "interface" of the object*
  - *There are, however, some conventions, i.e. rules that do not have an impeditive characteristic ([5]):*
    - prefix with an underscore the attributes or methods that we don't want to be public (_xpto)
      - *let's say they're members that in other languages we would classify as private or protected*
    - do not prefix our names with a double underscore (let's leave this possibility to the predefined names of the language)
      - *as you may have noticed, the names used internally by the system begin and end up double underscore (__xpto__), to serve as an alert*

        (these are keywords that should not be changed – they should not be used for another purpose)

The self clause in python classes

class

- *a class is defined with the keyword 'class'*
- *all its members (attributes and methods) must form an indented block*

self

- *serves for objects to self-reference themselves*
- *equivalent to C++, Java and C 'this', #*
- *will be decisive in differentiating object members (methods and attributes) from class members (static members)*
  - for now, just know that Python converts invocations of type obj.method(a1,a2) into X.method (obj,a1,a2), being obj an object of any class X.

  (calling a method from an instance with n arguments is equivalent to calling the corresponding function of the class with these n arguments preceded by a first with the instance reference – self)

Classes

- The simplest possible class is shown in the following example

```
class Person:
    pass  # An empty block

p = Person()
print(p)
```

Output:

```
<__main__.Person instance at 0x10171f518>
```

  - *in this case, the word 'pass' serves to indicate an empty block*
    - that will be the body of the Person class
  - *an object of the Person class is then created, and is then referenced by the variable p*
  - *the simple print of the object shows us: the current module (__main__), the class to which the object belongs (Person) and the address of its location in memory (0x10171f518)*

Methods

- Let's look at the self in action:

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

Output:

```
Hello, how are you?
```

  - *Note that the method say_hi() is invoked without any argument but, still, we have the self parameter in its definition*
    - is the inclusion of the self in the definition that causes the method to be of the object, not of the class

Initializing methods (constructors)

- Constructor methods in Python are named __init__

  (in Java, C++ and C# have the name of the class itself, as we know)

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

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
```

Output:

```
Hello, my name is Swaroop
```

Here, we define the __init__ with a name parameter (in addition to the self parameter)

- *but we also create a new attribute with the name 'name', using self as a qualifier*
  - is this qualifier that makes it an object attribute, not a class attribute

- In Python each class has only one constructor
  - *even so, different forms of initializations can be ensured through optional parameters*
- Therefore, there are only 2 types of constructors in Python
  - *default builder – one that has no parameters and is automatically generated whenever no other system is explicitly set*
  - *parameterized builder - all of which have parameters.*

Other special methods

- In addition __init__(self), there are other methods that have a special meaning within a Python class
  - *are typically used to emulate behaviors of certain functions or operations that have as operating the object*
    - For example, if we want to use obj[key] indexing with objects in our class (exactly as it is done in lists), then all we need to do is implement the __getitem__()

      (this is precisely what Python does in the list class)

- Below are some of the special methods
  - *For a more detailed knowledge of these and all other special methods, consult:*

  https://docs.python.org/3/reference/datamodel.html#special-method-names

Special methods

- `__init__(self, ...)`

  It is automatically executed at the time of object creation - known as constructor method.

- `__del__(self)`

  It automatically runs immediately before the object is destroyed - known as the destructor method.

- `__str__(self)`

  Invoked when using the print(obj) or str(obj) functions, being obj one of the objects in the class.

- `__lt__(self, other)`

  Invoked when used the smaller operator (obj<other, being obj and other objects of the class).

  Similarly, there are similar methods for all other operators (+, ==, >=, etc.)

- `__getitem__(self, key)`

  Invoked when obj[key] indexing is used, being obj one of the objects in the class.

- `__len__(self)`

  Invoked when using the len(obj) function, being obj one of the objects in the class

Identity vs Equality

- Consider the following objects:
  - *x1=[1,2,3]; x2=[1,2,3] ; x3=x1;*
- Two objects can be the same, but not with the same identity
  - *Example: x1 e x2*
- Already having the same identity, they are necessarily equal
  - *Example: x1 e x3*
- Let's see how these objects behave with the equality operator
  - *x1==x2*

    True
  - *x1==x3*

    True
  - *x2==x3*

    True

and with the identity operator

- *x1 is x2*

  False
- *x1 is x3*

  True
- *x2 is x3*

  False

- Therefore,
  - *the == operator is equivalent to the Java equals() method,*
  - *and the 'is' is equivalent to the == operator of Java*
- When using the equality operator, the method is __eq__ (self, other)
  - *That is, x1==x2 is the same as x1.__eq__(x2)*
  - *Therefore, if our classes are concerned, the == operator will only behave appropriately if we properly define the __eq__ (self, other)*

    (precisely the care we had to take with the equals() method of Java)

Iterators

- An iterator is an object that allows us to scroll through the elements of a collection or sequence, without having to know the internal organization of these structures.
- In Python, an iterator has the __next__(),
  - *that returns a different element from the collection each time it is called,*
  - *and throws the exception* StopIteration when there are no more elements
- This method and the exception is all we need to access the elements of the collection, one by one,
  - as illustrated in the example

(see below how to get the it iterator object)

```
lista=[2, 5, 1, 3]
it=iter(lista)
while True:
    try:
        x=next(it) #x=it.__next__()
        print(x, end=' ')
    except StopIteration:
        break
```

```
2 5 1 3
```

Iterable

- An iterable is an object (typically a collection or sequence) that contains the method __iter__(),
  - *which returns an iterator object*
- In the previous example, the statement iter(lista) invokes the method lista.__iter__()
- The iteration of an iterably collection or sequence is done in a more convenient way using the structure for
  - *Equivalent solution for the previous example, without the explicit use of iterators:*

```
lista=[2, 5, 1, 3]
for x in lista:
    print(x, end=' ')
```

```
2 5 1 3
```

  - *Note, however, that this more compact solution ends up being translated by Python into the corresponding explicit iteration, similar to the one shown earlier*
- The function range(), often used in the for loop, returns precisely an iterable object

```
for i in range(1,50,3):
    print(i, end=' ')
```

```
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
```

Making our objects iterable

- How to make a class iterable?
  - *A method is defined in this class __iter__(), that returns an iterator object*
    - that returns an object that contains the method __next()__
- In fact, in Python the iterator object itself includes, in addition to the __next()__, the __iter__()
  - *which also makes it an iterable object*
  - *being for that reason that one can use the for and the in, either in iterable objects, or on the iterators themselves*

  (the method __iter__() of an iterator is limited to returning self)
- Making a class iterable gives some work...
  - *involves, in addition to implementing the __iter__() in the iterable class, the creation of the iterator object and its method __next()__*

Generating functions

- Fortunately, Python provides us with an additional mechanism that greatly simplifies the creation of iterators – the generating functions
  - *A generating function returns a special object (the generating object) that behaves like an iterator (it is an iterator type)*
  - *In this function we only have to indicate which values we want to be returned sequentially by the generator object, one for each invocation of its method* __next__()
    - we then define the function as if it returned multiple values, one at a time (one for each method invocation __next__() of the generator object)
    - is through special instruction yield that we indicate each of the values to be returned (with the return statement only the first...)
  - *Thus, the simplest way to make a collection iterable is to implement its method* __iter__() *as a generating function.* ☺

Generate an iterator - example

- In the following example, the following method is implemented __iter__() of the class that is intended to be iterable (Students) as a generating function
  - *therefore returns a generator object, which is no longer an iterator*

```
class Alunos:
    def __init__(self, *nomes):
        self.nomes=nomes

    def __iter__(self):
        for nome in self.nomes:
            yield nome
```

- Being the class students iterable, we can already go through it with a cycle for....

```
alunos=Alunos('Ana','Rita','Rui')

for a in alunos:
    print(a)
```

```
Ana
Rita
Rui
```

solve exercise #6 and #7

from the book of exercises