Light on Python

November 8, 2015

Contents

1 Introduction

In this course:

- You'll learn Python the way a child would, even if you are an adult. Children are experts in learning. They learn by doing, and pick up words along the way. In this text the same approach is followed. Not everything is defined or even explained. Just try to find what makes the example code tick by guessing and experimenting. Regularly try to put together something yourself. Play with it. Evolution has selected playing as the preferred way of learning. I will not claim to improve on that.
- You'll be addressed like an adult, even if you are a child. Having a separate childs world populated by comic figures, Santa Claus and storks bringing babies is a recent notion. Before all that, it was quite normal to have twelve year old geniusses. But don't worry, programming can be pure fun, both for children and adults.
- You'll focus upon a very effective way of using Python right from the start. It is called Object Oriented Programming. And you'll learn some Functional Programming as well. Don't bother what these words mean. It'll become clear underway. There are also less important things to learn about Python. You can do so gradually if you wish, while using it. Just stay curious and look things up on the Internet.

I learned to program as a child, my father was programming the first computers in the early 1950's. We climbed through a window into the basement of the office building of his employer, a multinational oil company. Security was no issue then. Programming turned out to be fun indeed. And it still is, for me!

2 Objects

2.1 Your first program

Install Python 3.x. The Getting Started topic on www.python.org will tell you how. You will also need an editor. If you're on Windows, Google for Notepad++. If you're on Linux or Apple, you can use Gedit. Then run the following program:

```python
cities = ['Londen', 'Paris', 'New York', 'Berlin'] # Store 4 strings into a list
print ('Class is:', type (cities))               # Verify that it is indeed a list

print ('Before sorting:', cities)                # Print the unsorted list
cities.sort ()                                   # Sort the list
print ('After sorting: ', cities)                # Print the sorted list
```

Listing 1: prog/sort.py

The pieces of text at the end of each line, starting with #, are *comments*. Comments don't do anything, they just explain what's happening. But some explanation will help. 'London', 'Paris' and 'New York' are strings, pieces of text. You can recognize such pieces of text by the quotes around them. All four of them are *objects* of type *string*. Programmers would say these four objects are *instances* of *class* string, but they mean the same thing. To clarify, a particular dog is an instance of class Dog. There may be classes for which there are no instances. Class Dinosaur is such a class, since there are no (living) dinosaurs left. So a class in itself is merely a description of a certain category of objects.

Line 1 of the previous program is actually shorthand for line 1 of the following program:

```python
cities = list (('Londen', 'Paris', 'New York', 'Berlin')) # Construct list object from 'tuple' of 4 string objects
print ('Class is:', type (cities))                         # Verify that it is indeed a list

print ('Before sorting:', cities)                          # Print the unsorted list
cities.sort ()                                             # Sort the list
print ('After sorting: ', cities)                          # Print the sorted list
```

Listing 2: prog/sort2.py

So you construct objects of a certain class by using the name of that class, followed by (). Inside this () there maybe things used in constructing the object. In this case the object is of class *list*, and there's a so called *tuple* of cities inside the (). Since the tuple itself is also enclosed in (), you'll have *list ((...))*, as can be seen in the source code. A tuple is an immutable group of objects. So you could never sort a tuple itself. But the list you construct from it, is mutable, so you can sort it.

Once it works, try to make small alterations and watch what happens. Actually DO this, it willl speed up learning

2.2 Specifying your own classes

Generally, in a computer program you work with many different classes of objects: buttons and lists, images and texts, movies and music tracks, aliens and spaceships, chessboards and pawns.

So, looking at the "real" world: you are an instance of class *HumanBeing*. Your mother is also an instance of class *HumanBeing*. But the object under your table wagging its tail is an instance of class *Dog*. Objects can do things, often with other objects. You're mother and you can walk the dog. And your dog can bark, as dogs do.

Lets create a Dog class in Python, and then have some actual objects (dogs) of this class (species):

```python
class Dog:               # The species is called Dog
    def bark (self):     # Define that this dog itself can bark
        print ('Wraff!') # Which means saying "Wraff"


your_dog = Dog ()       # And than lets have an actual dog

your_dog.bark ()        # And make it bark
```

Listing 3: /prog/dog.py

Now lets allow different dogs to bark differenly by adding a *constructor* that puts a particular sound in a particular dog when it's instantiated (born), and then instantiate your neighbours dog as well:

```
class Dog:                      # Define the dog species
    def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
        self.sound = sound      # Stores accepted sound into self.sound field inside new dog

    def bark (self):            # Define bark method
        print (self.sound)      # Prints the self.sound field stored inside this dog

your_dog = Dog ('Wraff')        # Instantiate dog, provide sound "Wraff" to constructor
neighbours_dog = Dog ('Wooff')  # Instantiate dog, provide sound "Wooff" to constructor

your_dog.bark ()                # Prints "Wraff"
neighbours_dog.bark ()          # Prints "Wooff"
```

Listing 4: /prog/neighbours_dog.py

After running this program and again experimenting with small alterations, lets expand it further. You and your mother will walk your dog and the neighbours dog:

```
 1 class HumanBeing:               # Define the human species
 2     def walk (self, dog):       # The human itself walks the dog
 3         print ('\nLets go!')    # \n means start on new line
 4         dog.escape ()           # Just lets it escape
 5
 6 class Dog:                      # Define the dog species
 7     def __init__ (self, sound): # Constructor, named __init__, accepts provided sound
 8         self.sound = sound      # Stores accepted sound into self.sound field inside new dog
 9
10     def bark (self):            # Define bark method
11         print (self.sound)      # It prints the self.sound field stored inside this dog
12
13     def escape (self):          # Define escape method
14         print ('Run to tree')   # The dog will run to the nearest tree
15         self.bark ()            # It then calls upon its own bark method
16         self.bark ()            # And yet again
17
18 your_dog = Dog ('Wraff')        # Instantiate dog, provide sound "Wraff" to constructor
19 neighbours_dog = Dog ('Wooff')  # Instantiate dog, provide sound "Wooff" to constructor
20
21 you = HumanBeing ()             # Create yourself
22 mother = HumanBeing ()          # Create your mother
23
24 you.walk (your_dog)             # You walk your own dog
25 mother.walk (neighbours_dog)    # your mother walks the neighbours dog
```

Listing 5: prog/walking_the_dogs

Run the above program and make sure you understand every step of it. Add some print *statements* printing numbers, to find out in which order it's executed. Adding such print statements is a simple and effective method to *debug* a program (find out where it goes wrong).

In the last example the *walk* method, defined on line 2 receives two *parameters* (lumps of data) to do its job: *self* and *dog*. It then *calls* (activates) the *escape* method of that particular dog: *dog.escape*. Lets start following program execution from line 24 at statement *you.walk (your_dog)*. This results in calling (activating) the *walk* method with parameter *self* referring to object *you* and parameter *dog* referring to object *your_dog*. So the object before the dot in *you.walk (your_dog)* is passed to the *walk* method as parameter *self*, and *your_dog* is passed to the *walk* method as parameter *dog*.

Parameters used in calling a method, like *you* and *your_dog* in line 24 are called *actual parameters*. Parameters that are used in defining a method, like *self* and *dog* in line 2 are called *formal parameters*. The use of formal parameters is necessary since you cannot predict what the names of the actual parameters will be. In line 25, statement *mother.walk (neighbours_dog)* different actual parameters will be substituted for the formal parameters. Passing parameters to a method is a general way to transfer information to that method.

2.3 Indentation, capitals and the use of _

As can be seen from the listings, indentation is used to tell Python that something is part of what was above.

When you specify your own *classes*, it is comon practice to start them with a capital letter and use capitals on word boundaries: *HumanBeing*.

For *objects*, their *attributes* (which are also objects) and what they can do, their *methods*, in Python it is common to start with a lowercase letter and use _ on word boundaries: *bark, your_ dog.* If you want to learn a style that is consistent over multiple programming languages, use capitals on word boundaries for *objects* and *methods* as well, but start always start them with a lowercase letter.

By the way WritingClassNamesLikeThis or writingAllOtherNamesLikeThis is called Camel Case, while writing_all_other_names is called Pothole Case.

Constructors, the special methods that are used to *initialize* objects (give them their start values), are always named *__init__*.

3 Encapsulation

All objects of a certain class have the same attributes (data fields) but with distinct values, e.g. objects of class *Dog* have the attribute *self.sound*. And all objects of a certain class have the same methods (actions). For our class *Dog* in the last example, those are the methods *__init__*, *bark* and *escape*. Objects can have dozens or even hundreds of attributes and methods. In line 4 of the previous example, method *walk* of a particular instance of class *HumanBeing,* referred to as *self*, calls method *escape* of a particular instance of class *Dog*, referred to as *dog*. Note that uppercase letters and lowercase letters are considered distinct in Python. *Dog* is a class and *dog*, *your_ dog* and *neighbours_ dog* are particular dogs, so objects of that class.

So in the example *you.walk* calls *your_ dog.escape* and *mother.walk* calls *neighbours_ dog.escape*. Verify this by reading through the code step by step and do not proceed until you fully and thoroughly undersand this. If needed, print some numers

In general any object can call any method of any other object. And it also can access any attribute of any other object. So objects are highly dependent upon eachother. That may become a problem. Suppose change your program, e.g. by renaming a method. Then all other objects that used to call this method by its old name will not work anymore. And changing a name is just simple. You may also remove formal parameters, change their meaning, or remove a method alltogether. In general, in a changing world, you may change your design. As your program grows bigger and bigger, the impact of changing anything becomes disastrous.

To limit the impact of changing a design, standardisation is the answer. Suppose we have two subclasses of HumanBeing, NatureLover and CouchPotato. Objects of class NatureLover go out with their dogs to enjoy a walk. Objecs of class CouchPotato just deliberately let the dog escape from the doorstep, that it might walk itself while they're watching their favorite soap.