

# A BRIEF INTRODUCTION TO PYTHON

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

- licensed with the standard “[Creative Commons Attribution-ShareAlike 4.0 International](#)”,  
which allows changes and sharing of your content.

# Object and class attributes

- Attributes are variables that are 'confined' to the 'namespace' of a class or object
  - *class attributes are shared by all objects in the class - there is only one copy of this variable*
  - *Object attributes are private to each individual instance of a class - each object has its own copy of that attribute. See the following example:*

# Object and class members - example

- **population** belongs to the Robot class and is therefore a class attribute.
- The attribute **name** belongs to the object, as it is defined with the `self` qualifier
- The two attributes are now referenced with different qualifiers: `self.name` and `Robot.population`.
- **how\_many()** is a class method, since it has as its first argument 'cls' (class abbreviation) and is preceded by the decorator `@classmethod`

(for more details see [3])

```
class Robot:  
    """Represents a robot, with a name."""  
  
    # A class variable, counting the number of robots  
    population = 0  
  
    def __init__(self, name):  
        """Initializes the data."""  
        self.name = name  
        print("(Initializing {})".format(self.name))  
        # When this person is created, the robot  
        # adds to the population  
        Robot.population += 1  
  
    def die(self):  
        """I am dying."""  
        print("{} is being destroyed!".format(self.name))  
        Robot.population -= 1  
        if Robot.population == 0:  
            print("{} was the last one.".format(self.name))  
        else:  
            print("There are still {} robots working.".format(  
                  Robot.population))  
  
    def say_hi(self):  
        print("Greetings, my masters call me {}.".format(self.name))  
  
    @classmethod  
    def how_many(cls):  
        print("We have {:d} robots.".format(cls.population))
```

```

droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()

```

In Python, class method and method static are not quite the same thing:

- a class method has the parameter `cls`
  - *that references the class itself*
  - *and is preceded by @classmethod*
- already a static method, has neither the parameter `cls` nor the `self` (thus not having access to either the reference of the class or the object)
  - *it will be just a utilitarian function that operates solely on its own parameters (and that will have some logical relationship with the class)*
  - *and will have no need to deal with instances of the class itself*
  - *is preceded by @staticmethod*

# Test

- In this example, we saw the use of docstrings, both in the methods and in the class itself
  - *we can then access the class documentation with `Robot.__doc__`*
  - *and access the documentation of the method with `Robot.sayHi.__doc__`*

## Output:

```

(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.

(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.

We have 0 robots.

```

# Python Inheritance

- In Python, the class `object` is the 'mother' of all classes
  - *Like Java and C#, all classes derive from the object class, either directly (when not explicitly derived from another) or through their superclasses,*
  - *thus ensuring some basic functionality in any class that is created*
- Multiple Inheritance is allowed between classes
- To create a class by derivation of another, you place yourself in front of the name of the class that is being created the name of the base class, in curved parentheses
  - `class SubClass(ClasseBase): ...`
- The constructor of a subclass, when defined, must invoke the constructor of the base class
  - *If no constructor is defined in the subclass, then, and only in this situation, it can be considered that the constructor of the base class is inherited*
- From a subclass, there are two ways to invoke the constructor of its base class:
  - `super().__init__([arguments])`
  - `ClasseBase.__init__(self, [arguments])`
- Because, as a rule, each class is defined in a separate file, the definition of a subclass must be preceded by the import of its base class

```
from moduloDaClasseBase import ClasseBase
class SubClasse(ClasseBase): ...
```

```

class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('Initialized SchoolMember: {}'.format(self.name))
    def tell(self):
        '''Tell my details.'''
        print('Name:{} Age:{}'.format(self.name, self.age), end=" ")
class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age) #super().__init__(name, age)
        self.salary = salary
        print('Initialized Teacher: {}'.format(self.name))
    def tell(self):
        SchoolMember.tell(self) # or super().tell()
        print('Salary: {}'.format(self.salary))
class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age) #super().__init__(name, age)
        self.marks = marks
        print('Initialized Student: {}'.format(self.name))
    def tell(self):
        SchoolMember.tell(self) # or super().tell()
        print('Marks: {}'.format(self.marks))

```

# Inheritance/Polymorphism

In Python, all methods are virtual

- and since what we have are always references to objects, polymorphism is guaranteed...

```

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()

```

Output:

```

(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary:"30000"
Name:"Swaroop" Age:"25" Marks: "75"

```

# Files

```
poem = '''\nProgramming is fun\nWhen the work is done\nif you wanna make your work also fun:\n    use Python!\n\n# Open for 'w'riting\nf = open('poem.txt', 'w')\nf.write(poem) # Write text to file\nf.close() # Close the file\n\n# If no mode is specified,\n# 'r'ead mode is assumed by default\nf = open('poem.txt')\nwhile True:\n    line = f.readline()\n    # Zero length indicates EOF\n    if len(line) == 0:\n        break\n    # The `line` already has a newline\n    # at the end of each line\n    # since it is reading from a file.\n    print(line, end=''\n\nf.close() # close the file
```

- You can open files and write and read in them, creating a file class object, through the `open()` function, and using the `read()`, `readline()` and `write()` methods
  - *at the end it ends with the method `close()`*
  - *In opening mode we can specify whether we will read ('r'), write ('w') or add ('a'), in text mode ('t') or in binary mode ('b'), among other options*
    - by default, the file is considered text and is opened in read mode

## Output:

```
Programming is fun\nWhen the work is done\nif you wanna make your work also fun:\n    use Python!
```

# Make objects persistent

- Python provides us with a very simple way to write an object to a file – making it persistent – to make it persistent, therefore.
  - *Just use the standard module called pickle, as illustrated in the example.*

```
import pickle

# The name of the file where we will store the object
shoplistfile = 'shoplist.data'
# The list of things to buy
shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = open(shoplistfile, 'wb')
# Dump the object to a file
pickle.dump(shoplist, f)
f.close()

# Destroy the shoplist variable
del shoplist

# Read back from the storage
f = open(shoplistfile, 'rb')
# Load the object from the file
storedlist = pickle.load(f)
print(storedlist)
f.close()
```

Output:

```
['apple', 'mango', 'carrot']
```

# Python Standard Library

- The Python Standard Library contains a vast number of useful modules and comes in all Python installations.
  - *It is important to know her, since with her many problems can be solved quickly.*
- Let's illustrate only one of the most used modules in the library (Module [sys](#)), but a wide variety of other modules can be found and used,
  - as will be the case, for example, with regular expressions ([re](#)), of the command line options ([argparse](#)), of date/time types ([datetime](#)) and debugging ([pdb](#)).
  - Detailed documentation of all python standard library modules can be found at the following address:  
<https://docs.python.org/3/library/>

# The sys module

- The sys module contains system-specific features
  - We have seen before that this module integrates the list `sys.argv`, which contains the arguments of the invocation line of the running script.
- Suppose now that you want to check if the installed version of Python is the 3
  - The sys module gives us that information:

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=0, releaselevel='final', serial=0)
>>> sys.version_info.major == 3
True
```

# Lists by Understanding

- Comprehension Lists allow us to obtain a new list from an existing one (or, more specifically, from an existing iterable object),
  - *using the 'for' to scroll through the original list,*
  - *and indicating the operation to be performed on each element of the original list*
- This is a fast and very compact way to build lists
- Suppose we have a list of numbers and we intend to get a new list with all these numbers multiplied by two:

```
lista1=[2,3,4]
lista2=[2*x for x in lista1]
print(lista2)
```

```
[4, 6, 8]
```

Note that 'list1' does not need to be a list; it is enough to be an iterable object

# Lists by Understanding

- What if the idea was to get the values doubled only for numbers greater than 2?
  - *at that time we applied a filter (if...) to the List by Understanding:*

```
lista1=[2,3,4]
lista2=[2*x for x in lista1 if x>2]
print(lista2)
```

```
[6, 8]
```

- Syntax remembers the notation used in set theory:
$$\{2x : x \in \{2, 3, 4\} \wedge x > 2\}$$
- In essence, The Comprehension Lists carry out 3 types of operations:
  - *select values from an iteration,*
  - *apply a transformation to each of these values,*
  - *and form a new list with these results.*

# Generating expressions

- We simply exchange, in a list by understanding, the straight parentheses for curved, to obtain, not a list, but a generating object, capable of iteratively reproducing (were it if it were not an iterator) the resulting values
  - *this new expression is designated Generating Expression*

```
▶ M↓  
it=(2*x for x in lista1 if x>2)  
print(it)  
  
<generator object <genexpr> at 0x000002E9BC54A6D0>  
  
▶ M↓  
next(it)  
6  
  
▶ M↓  
next(it)  
8  
  
▶ M↓  
next(it)  
  
-----  
StopIteration                                     Traceback (most recent call last)  
<ipython-input-16-bc1ab118995a> in <module>  
----> 1 next(it)  
  
StopIteration:
```

# Lambda function

- A lambda function is an anonymous one-line function
  - *It is an additional procedural abstraction mechanism that Python supports, like other languages.*
- Its formulation is simple:

`lambda <formal parameters>: <expression>`

- *syntactically, boils down to a simple expression*
- *is set using the prefix `lambda`, instead of the `def`*
- *is anonymous because it doesn't have an associated name*
- *does not need the `return` statement (the `return` of the result is implied)*
- *may contain neither control instructions nor local variables, other than the parameters*
- *can be used whenever a function object is expected*

# Lambda function

- These are usually small functions that are used
  - *when they only need to be executed once (they are set at the time we use them),*
  - *and syntactically the conventional definition of the function is not adequate*
- The purpose of the lambda function is to produce, in certain contexts, better quality code
  - *However, it will always be possible to implement the solutions without resorting to this type of*  
*(Basically, what you get with these types of functions is to improve the style of the code)*
- Example of lambda function:

```
(lambda x,y: x.lower()==y.lower())('ana','Ana')
```

```
True
```

# Lambda function with map()/filter()

- The usefulness of the lambda function becomes more evident when combined with other functions
  - as is the case with the "functions" map() and filter()

map(function, list):

- aims to apply a function to all elements of a list, resulting in an iterator (map class object) for its results

```
lista1=[2,3,4]
lista2=list(map(lambda x: 2*x, lista1))
print(lista2)
```

```
[4, 6, 8]
```

filter(function, list):

- aims to apply a filter (defined by a Boolean function) to all elements of a list, resulting in an iterator (object of the filter class) for its results

```
lista1=[2,3,4]
lista2=list(filter(lambda x: x%2==0, lista1))
print(lista2)
```

```
[2, 4]
```

solve **exercise #8**

from the book of exercises