Python for Tooling

Prepared By

Ahmed Magdy

Agenda

- Introduction to Tooling
- Intro to Python
- Data Types
- Variables
- Strings
- Lists / Tuples
- Loops and Conditional Statements
- Methods and Global Variables
- Classes
- Python Boilerblate

Introduction to Tooling

Introduction to Tooling

- It is making a piece of code (Tool) to be used in making some repeated or hard or long processing on one or more files.

  i.e. Creating, Writing and Modifying Excel files in different computer directories.

- Benefits:
  1. Save Time.
  2. Higher accuracy and efficiency.

Intro to Python

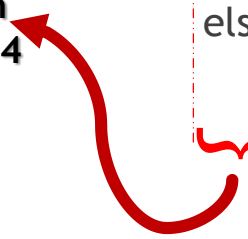
Intro to Python

- Scripting language which is translated line by line using the interpreter.
- Case sensitive.
- Has many versions (In this sessions Python 2.7 is to be discussed).
- Can be attached with a user interfacing.
- Python is a programming language that lets you work quickly and integrate systems more effectively.
- Has ability to communicate with the OS and do some operations based on communications with OS.
- No brackets or Semicolon in Python like C language , instead we use indentation as :-

i.e.

```
if name == 'ITI' :
        print "ITI_Ismailia"
else:
        print "Not ITI"
```

This Indentation Is typically 2 to 4 spaces



Intro to Python

- Writing Comments:

  “”” This would be a multiline comment

  in Python that spans several lines and

  describes your code, your day, or anything you want it to

  “””

  Or

  # For commenting just one line not multi

Install Environment

Install Environment

- Download the .zip folder from the following links then download the packages inside it :

  https://drive.google.com/open?id=164ailcwRIECdDnHu6xuOZdYkBnDt0DEf

Data Types

Data Types

- Boolean : True or False
- Integer
- String
- Float
- Complex (i.e. complex numbers represented in real and imaginary -->7 + 65J)
- List
- Tuple

Data Types

- Where all the previous types can be used in casting variables directly as the following example :

```
A=5                     #  This is a variable A
print str(A)+”ITI”      # Using the “print” function to print the string-casted
                          variable A in addition to the word “ITI”
```

Lab 1

Lab 1

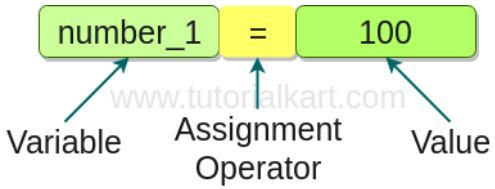
- Download the Python module file from the following link and run it into the cmd using

python NameOfFile.py

https://drive.google.com/open?id=1-grmNrGhu_LEDQTwbW70H37vmFnsZHHT

Variables

Variables



- No declaration to variables in Python ☺
- Typing the name of the variable in the left hand side and the value on the right side implicitly defines the variable with its type

i.e. x=5 # An integer variable is implicitly defined because it equals integer value.

y= “ITI” # A string variable is automatically defined as the variable name on the left side equals “string” value on the right side.

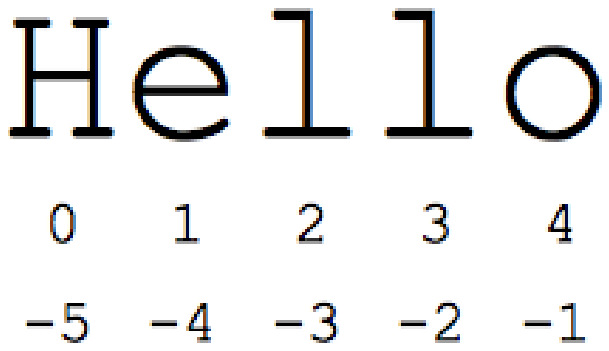
Strings

Strings

- String literals can be enclosed by either double or single quotes, although single quotes are more commonly used. *i.e. var= “ITI“*
- Python strings are "immutable" which means they cannot be changed after they are created.
- **String Concatenation:-**

```
s = 'hi’            ## Defining a string variable
print s[1]          ## output is  “I”
print len(s)        ## output is  “2”
print s + ' there'  ## output is  “hi there” due to concatination
```

Strings



- **Raw Operator:**

  A "raw" string literal is prefixed by an 'r' and passes all the chars through without special treatment of backslashes.

  i.e. rawVar = **r**' this\t\n and that '

  print rawVar ## Output is "this\t\n and that"

- **String Slicing:-**

  The "slice" syntax is a handy way to refer to sub-parts of sequences where :-

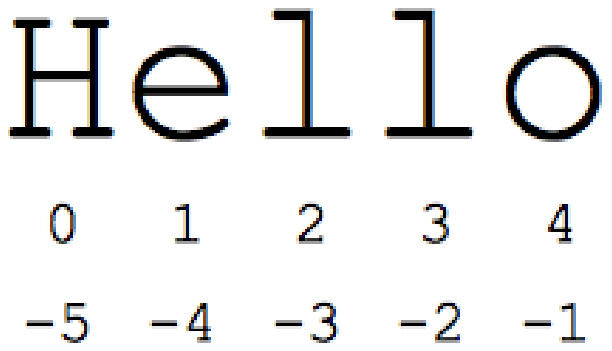
  s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4

  s[1:] is 'ello' -- omitting either index defaults to the start or end of the string

  s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)

  s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.

Strings



- **String Methods:**
  1. s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
  2. s.strip() -- returns a string with whitespace removed from the start and end
  3. s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
  4. s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
  5. s.split('delim') -- returns a list of substrings separated by the given delimiter.
  6. s.join(list) -- opposite of split(), joins the elements in the given list together

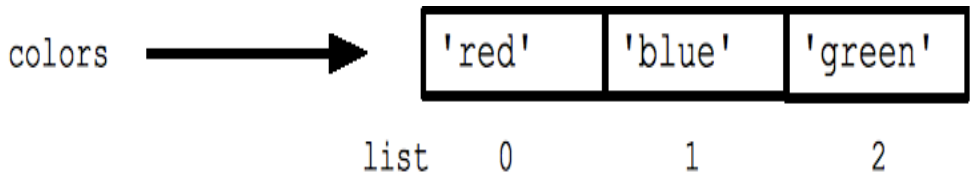
Lists / Tuples

Lists / Tuples

- **Lists:**
  1. Like Arrays in C language.
  2. List literals are written within square brackets [ ].
  3. Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.

i.e. colors = ['red', 'blue', 'green']

```
print colors[0]    ## red
print colors[2]    ## green
print len(colors)  ## 3
```



Lists / Tuples

- **Lists:**

  List Methods:

  1. list.append(elem) -- adds a single element to the end of the list.
  2. list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
  3. list.index(elem) -- searches for the given element from the start of the list and returns its index.
  4. list.reverse() -- reverses the list in place (does not return it)

Lists / Tuples

- **Lists:**

  Copying list is not made just by equal sign !

  Copying an array or a list can not be done using the equal operator “=” as this do only a shallow

  copy, but to make deep copy (I mean the source and the copy have different versions in two different memory locations) we should use list() :-

Lists / Tuples

- **Lists:**

```
>>> my_list=['1','2','4']
>>> new_list=my_list
>>> id(my_list)
41149896L
>>> id(new_list)
41149896L
>>> my_list[0]='7'
>>> print my_list
['7', '2', '4']
>>> id(my_list)
41149896L
>>> print new_list
['7', '2', '4']
>>> id(new_list)
41149896L
>>> new_list = list(my_list)
>>> my_list[0]='9'
>>> print new_list
['7', '2', '4']
>>> print my_list
['9', '2', '4']
```

Lists / Tuples

- Tuples:
  - Tuples are a group of values like a list and are manipulated in similar ways. But, tuples are fixed in size once they are assigned.
  - Tuples are defined by parenthesis ():

    i.e. myGroup = ('Rhino', 'Grasshopper', 'Flamingo', 'Bongo')
  - Some advantages of tuples over lists:
    1. Elements to a tuple. Tuples have no append or extend method.
    2. Elements cannot be removed from a tuple.

i.e. tuple = (1, 2, 'hi')

tuple[2] = 'bye' ## NO, tuples cannot be changed

Lab 2

Lab 2

- Download the Python module file from the following link and solve it

https://drive.google.com/open?id=13piVzr9-trtRDDZ-x_iw2XCqjLckBVKN

Loops and Conditional Statements

Loops and Conditional Statements

- **If Statement :**

  Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements.

```
if speed >= 80:
    print 'License and registration please’
    if mood == 'terrible' or speed >= 100:
        print 'You have the right to remain silent.'
    elif mood == 'bad' or speed >= 90:
        print "I'm going to have to write you a ticket."
        write_ticket()
 else:
    print "Let's try to keep it under 80 ok?"
```

Loops and Conditional Statements

- FOR and IN:

  The *for* construct -- for var in list -- is an easy way to look at each element in a list (or other collection).

  i.e.

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
    sum += num
print sum  ## 30
```

Loops and Conditional Statements

- FOR and Range:

  The range(n) function yields the numbers 0, 1, ... n-1

  i.e.

```
## print the numbers from 0 through 99
for num in range(100):
    print num
```

```
## print the numbers from 1 to 5
for num in range(1,6):
    print num
```

Loops and Conditional Statements

- While Loop:

  Python also has the standard while-loop, and the *break* and *continue* statements .

i.e.

```
## Access every 3rd element in a list
i = 0
while i < len(a):
      print a[ i ]
       i = i + 3
```

Lab 3

Lab 3

- Download the Python module file from the following link and solve it

https://drive.google.com/open?id=1ZbllD8UpdGmh2ELS0iZMxoepQndKUrC-

Methods and Global Variables

Methods and Global Variables

- def Method_Name (Input_parameters) :

  ----

  return Variable or List or Tuple

} **Take care of the indentation**

- Defining a variable outside the scope of the method then redefining it inside the scope of the method does not mean that they are the same variable !!!!!

Methods and Global Variables

- Xspeed=20

```
def Method_Name ( Input_parameters ) :
        Xspeed="False Speed"
        print Xspeed    # Output is "False Speed"
print Xspeed    # Output is "20" as this is the
                        Global variable
```

Xspeed outside the method is different from the Xspeed inside the method.

Methods and Global Variables

- Xspeed=20

```
def Method_Name ( Input_parameters ) :
    global Xspeed
    Xspeed="False Speed"
      print Xspeed   # Output is "False Speed"
print Xspeed   # Output is "False Speed" as this is the Global variable which is
                 used as a global variable inside the method and modified
```

Now, Xspeed outside the method is the same as the Xspeed inside the method.

Lab 4

Lab 4

- Download the Python module file from the following link and solve it

https://drive.google.com/open?id=1KNogOaqXbGiRdDWx9JFWLNo0fs44bMP4

Classes

Classes

```
class NameOfClass :
    def __init__(self, name, salary= 3000) :
        self.name = name
        self.salary = salary
        self.empCount += 1

    def Class_First_Method(self) :
        print "Total Employee %d" % self.empCount
```

Defining the name of the class

The constructor method of the class which is called automatically when an instance of the class is declared and passing to it the needed input parameters. Also note the default value for "Salary" is 3000 so it may not be given in declaration of instance.

Classes

```
ClassType_Var = NameOfClass("Ahmed")      # Defining an instance of the class in the previous slide

ClassType_Var. Class_First_Method()       # Calling "Class_First_Method" method of the class "NameOfClass"
```

Python Boilerblate

Python Boilerblate

In normal case if we write our Python code without being inside a Method or Class will be interpreted normally, but what if our code fully written in the form of Method and class, then which thing will control which method to be invoked before another one.

Python Boilerblate

```
import time
```

Libraries to be imported

```
def mainMethodinPython():
    """ Main entry point of the app """
    print "hello world“ + str (time.time())


if __name__ == "__main__":
    """ This is executed when run from the command line """
    mainMethodinPython()
```

When a Python file is run directly, the special variable "__name__" is set to "__main__". Therefore, it's common to have the boilerplate if __name__ ==... shown above to call a main() function when the module is run directly, but not when the module is imported by some other module.

Any Questions or Comments ?

References

References

- Google's Python Class