What is Python?

Python is a quick and light interpreted scripting language that you can use to quickly prototype a tool or a project.

You can use it for everything from simple automation scripts to even full applications, neural networks, scientific computing, plots, graphical user interfaces, web back-ends, etc ...

From tutorialspoint:

> Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Programming in python

Opening up the python interpreter

```
ThisBeSilver:2017 - Python workshop diogoaguiam$
python
Python 2.7.12 |Anaconda 4.2.0 (x86_64)| (default, Jul
2 2016, 17:43:17)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM
build 2336.11.00)] on darwin
Type "help", "copyright", "credits" or "license" for
more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and
```

```
https://anaconda.org
>>>
```

Python Syntax

```
>>> 1+3
4
>>> a = 3
>>> a + 6
9
>>> b = "Hello world"
>>> b
'Hello world'
>>> print b
Hello world
>>> a = 3 # This is a comment
>>> a
3
>>> """ This is another comment """
' This is another comment '
>>> """ It can be multiline
... like this """
' It can be multiline \nlike this '
>>>
```

> ***Question:*** How can you define a multi line string variable?

Python can handle automatically the data formats: int, floats, complex, hex

```
>>> 1
1
>>> 1.0
1.0
```

```
>>> 32
32
>>> 3/2
1
>>> 10
10
>>> 1.2
1.2
>>> 0x30
48
>>> 1.2783468723e23
1.2783468723e+23
>>>
```

Mathematical operators: + - * / % **

```
>>> 1+2
3
>>> 2-1
1
>>> 2*2
4
>>> 10/5
2
>>> 10/3
3
>>> 10/3.0
3.3333333333333335
>>> 2**8
256
>>> 2**10
1024
>>> 2**10.1
1097.496025637164
>>> 2**10.
1024.0
>>> 0+1j + 2-3j
```

```
(2-2j)
>>>
```

Data types

These are the default data types in the base module of python.

Strings

Strings are a sequence of characters, like other programming languages, and can be indexed similarly.

```
>>> a = 'Hello World'
>>> print a
Hello World
```

Indexing strings

```
>>> a[0]
'H'
>>> a[10]
'd'
>>> a[-1]
'd'
>>> len(a)
11
>>> a[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

> ***Question:*** What is the value of a[11]?

> ***Note:*** The python interpreter executes each command or script line consecutively and raises an Error as soon as one is executed. This enables a quick and descriptive debugging of the script.

Additionaly, a subset of strings, arrays and lists in python can be indexed with a sequential range.

```
>>> a[0:3]
'Hel'
>>> a[:3]
'Hel'
>>> a[3:]
'lo World'
>>> a[:]
'Hello World'
>>> a[:-1]
'Hello Worl'
>>> a[:-2]
'Hello Wor'
>>> a[:-3]
'Hello Wo'
>>> a[-3]
'r'
>>> a[-3:]
'rld'
>>>
```

> ***Question:*** What is the value of a[:]?

Concatenating strings together

```
>>> b = '!!!'
>>> print a + b
Hello World!!!
>>> print a,b
```

```
Hello World !!!
>>>
```

> ***Question:*** How can you assign a new variable new that joins strings a and b?

Handling strings with numbers

```
>>> c = a + 22
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> c = a + str(22)
>>> c
'Hello World22'
>>>
```

> ***Question:*** What if you want to join string and number variables?

Strings can be formatted with variable info, similarly to other langauges.

```
>>> name = 'Dexter'
>>> age = 15
>>> print 'My name is %s and my age is %d'%(name,age)
My name is Dexter and my age is 15
>>>
```

```
>>> pi = 3.1415927
>>> print 'Pi is %f'%(pi)
Pi is 3.141593
>>> print 'Pi is %0.2f'%(pi)
Pi is 3.14
```

```
>>>
```

Lists

Lists are limited by the square brackets [,] brackets and are not limited by a single data type.

```
>>> [1, 2, 3]
[1, 2, 3]
>>> [1, 'a', 2.0]
[1, 'a', 2.0]
>>>
```

The indexing of lists is the same as for strings

```
>>> a = [0, 1, 2, 'a','string',3.14]
>>> a[:3]
[0, 1, 2]
>>> a[:-2]
[0, 1, 2, 'a']
>>>
```

> ***Question:*** What is the value of a[3]?

Methods related to lists:

Adding or removing elements to a list

```
>>> a = [0, 1, 2, 'a','string',3.14]
>>> a
[0, 1, 2, 'a', 'string', 3.14]
>>> a.append('Added to last')
```

```
>>> a
[0, 1, 2, 'a', 'string', 3.14, 'Added to last']
>>> a.pop()
'Added to last'
>>> a
[0, 1, 2, 'a', 'string', 3.14]
>>> a.insert(0,'New first element')
>>> a
['New first element', 0, 1, 2, 'a', 'string', 3.14]
>>>
```

Sorting lists

```
>>> a
['New first element', 0, 1, 2, 'a', 'string', 3.14]
>>> a.sort()
>>> a
[0, 1, 2, 3.14, 'New first element', 'a', 'string']
>>> a.reverse()
>>> a
['string', 'a', 'New first element', 3.14, 2, 1, 0]
>>>
```

Sorting lists with methods changes the underlying list. Better to use sorted to provide a sorted copy of the list:

```
>>> mylist = [4,2,1,2.3,10]
>>> mylist.sort()
>>> mylist
[1, 2, 2.3, 4, 10]
>>> mylist = [4,2,1,2.3,10]
>>> sorted(mylist)
[1, 2, 2.3, 4, 10]
>>> mylist
```

```
[4, 2, 1, 2.3, 10]
>>> sorted(mylist, reverse=True)
[10, 4, 2.3, 2, 1]
>>>
```

Extra: Custom sorting

```
>>> a = ['222','aaaaa','2']
>>> a
['222', 'aaaaa', '2']
>>> sorted(a,key=len)
['2', '222', 'aaaaa']
>>>
```

Create a list integer sequence

Usage: range(start, [stop, [step]])

```
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a = range(2,10)
>>> a
[2, 3, 4, 5, 6, 7, 8, 9]
>>> a = range(2,10,3)
>>> a
[2, 5, 8]
>>>
>>> a = range(2,10,0.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer step argument expected,
got float.
>>>
```

> ***Question:*** Why is there no 9 or in the result of range(2,10,3)?

> ***Note:*** The range generator is useful for loop sequences or generate integer arrays. However, for real data processing it is better to use a more robust library such as NumPy, which is explained later.

For floats we have to use another library (numpy.linspace)

Tuples

Tuples are defined in python with normal parenthesis (). Tuples are the same as lists, that they are a data sequenece with any data type. The main difference is that tuples are immutable, meaning that they can not be changed after being defined.

```
>>> b = [1,2,'aa']
>>> b
[1, 2, 'aa']
>>> b[0]='dd'
>>> b
['dd', 2, 'aa']
>>>
```

But with tuples we get:

```
>>> b = (1,2,'aa')
>>> b
(1, 2, 'aa')
>>> b[0]
1
>>> b[0] = 'dd'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
>>>
```

Dictionaries

Dictionaries are similar to lists but are indexed by keys, instead of integer indexes. Data stored in dictionaries is hashed, resulting in a much quicker lookup than using lists.

Usage: mydictiony = {'key_name': values}

Very efficient way to create a database and access its elements. Useful for configuration files, etc

```
>>> user = {}
>>> user
{}
>>> user['name'] = 'Dexter'
>>> user['age'] = 15
>>> user
{'age': 15, 'name': 'Dexter'}
```

> ***Question:*** How can we create command to print The user Dexter is 15 years old ?

```
>>> user['mylist'] = ['element1',2222,'string']
>>> user
{'age': 15, 'mylist': ['element1', 2222, 'string'],
'name': 'Dexter'}
>>> user['age']
15
>>> user['name']
```

```
'Dexter'
>>> user['name'] = 'Deedee'
>>> user['name']
'Deedee'
>>>
```

> ***Question:*** What would be **current** the output if we executed the command of the previous question?

Methods to handle dictionaries

```
>>> user.keys()
['age', 'mylist', 'name']
>>> user.values()
[15, ['element1', 2222, 'string'], 'Deedee']
>>>
```

Control flow conditions

Let's do some conditions now

Comparison operators:

- < ... Less than
- > ... More than
- <= ... Less than or equal
- >= ... More than or equal
- == ... Equal to
- <> ... Different
- != ... Different
- is ... Equality between objects
- is not ... Inequality between objects
- in ... Is contained in list

- not in ... Is not contained in

always return a True or False

```
>>> a = 1
>>> a == 1
True
>>> a != 1
False
>>> a >= 1
True
>>> b = 2
>>> a == 1 and b == 2
True
>>> a == 1 and b != 1
True
```

> ***Question:*** What is the output of 'abc' <> 'Abc'?

> ***Question:*** What is the output of 2 in [1,2,3,4]?

> ***Question:*** What about 2.0 in [1,2,3,4]?

> ***Question:*** And '2' in [1,2,3,4]?

> ***Question:*** And 'a' in 'abcd?

if control

```
if <condition 1> :
    <instruction 1>
elif <condition 2> :
    <instruction 2>
...

else :
```

```
    <default instruction>
```

> ***Note:*** Code blocks are are distinguished by their identations, instead of the more common curly brackets {} in other programming languages.

Example

```
>>> x = 1
>>> if x == 1:
...     print 'x is 1'
... else:
...     print 'x is not 1'
...
x is 1
>>>
```

What about checking if the name you are asking is inside a valid name list?

```
>>> namelist = ['John', 'Maria', 'Mohammed']
>>> user = 'George'
>>> if user in namelist:
...     print 'user', user, 'is in the name list'
... else:
...     print 'user',user, 'is not in the name list!'
...
user George is not in the name list!
>>>
```

> ***Note:*** Best practice in python uses 4 spaces instead of a tab character. A single tab character may not be equal to the same amount of space characters in different IDEs. If the separation

> indentation uses both tabs and spaces, the interpreter may raise an identation error. Some IDEs already convert tabs to 4 spaces. You should configure your IDE to do the same when programming in python.

```
>>> if True:
...     2
...     3
  File "<stdin>", line 3
    3
    ^
IndentationError: unindent does not match any outer
indentation level
>>>
```

Loop sequences

while

The while control sequence works similiar to other languages. The <instructions> are executed while <condition> is verified. When the condition is finished, the <else instructions> are executed.

```
while <condition>:
    <instructions>

else:
    <else instructions>
```

Example:

```
>>> i = 0
```

```
>>> while i<=5 :
...     print 'Number:',i
...     i += 1
...
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
>>>
```

for

The for control works a bit different in python.

For each iteration of for, the <variable> takes one of the values in the <sequence or list>. As soon as there are no more values in the list, the for loop stops.

```
for <variable> in <sequence or list >:
    <instructions>
else:
    <else instructions>
```

```
>>> for i in range(5):
...     print 'Number',i
... else:
...     print 'Loop finished!'
...
Number 0
Number 1
Number 2
Number 3
```

```
Number 4
Loop finished!
>>>
```

Control flow tools break, continue, pass

The control flow tools break, continue, and pass also work within for and while.

- break ... Stops the for or while loop sequence
- continue ... Skips the remaining code in the current loop iteration and moves on to the next iteration
- pass ... Does nothing. It is used solely for python syntax purposes, mainly to maintain identation

Example

```
for i in range(15):
    print 'Iteration %d'%(i)
    if i == 1:
        print 'pass'
        pass
    elif i == 2:
        print 'continue'
        continue
    elif i == 4:
        print 'break'
        break
    elif i == 5:
        print 'reached iteration 5'
    print 'End of iteration %d'%(i)
```

Result

```
Iteration 0
End of iteration 0
Iteration 1
pass
End of iteration 1
Iteration 2
continue
Iteration 3
End of iteration 3
Iteration 4
break
>>>
```

Nested code blocks

Nesting a control sequence or code block inside an existing code block is as simple as adding another identation level for the child code block.

```
>>> for i in range(3):
...     for j in range(2):
...         print 'Numbers: %d-%d'%(i,j)
...
Numbers: 0-0
Numbers: 0-1
Numbers: 1-0
Numbers: 1-1
Numbers: 2-0
Numbers: 2-1
>>>
```

Defining functions

Functions are essential part of any programming. Functions are

reproducible and act on the passed arguments.

In python functions are defined as blocks of code whose syntax is

```
def function_name(<argument1>, <argument2>):
    """ Optional but recomended description of the
function
    including input arguments
    and outputs
    """

    <instructions>
    return
```

An example function is

```
>>> def print_text(text="Hello World"):
...     print text
...     return
...
>>> print_text()
Hello World
>>> print_text("this is another text")
this is another text
>>> print_text(text="this is yet another text")
this is yet another text
>>>
```

A particular feature of python is that you can set the default value for the argument directly when defining the arguments. If the argument is not defined when executing the function, then the default value is used.

Multiple arguments may defined in a single function in the same line or in multiple lines

```
>>> def print_text( text1="Hello",
...                 text2="World"):
...     print text1,text2
...     return
...
>>> print_text()
Hello World
>>> print_text("Hey")
Hey World
>>> print_text("Hey","Earth")
Hey Earth
>>> print_text(text2="Earth",text1="What's up")
What's up Earth
>>> print_text("Earth",text1="What's up")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_text() got multiple values for
keyword argument 'text1'
>>>
```

> ***Note:*** Good python programming style practices allow arguments to be defined in the same line when total line length is not exagerated (<80 characters). Also, keep default value assignments together with no space, while separating arguments by a comma + space.

```
def function(arg1=1, arg2=2.0):
    return
```

Creating python scripts

Python is rarely meant to be programmed directly in the interpreter. Python code stored in python scripts ending with .py which are then executed by calling the interpreter with the script name passed as argument.

```
python script.py
```

script.py is simply:

```
print 'This code was executed from a python script'
```

```
$ python script.py
This code was executed from a python script
$
```

Boilerplate code for a python script

The boilerplate code in code/boilerplate.py is recomended as the starting structure for a python script.

```
#!/usr/bin/env python
""" Boilerplate description of the script
"""

def main():
    print "Hello world"

if __name__ == "__main__":
    main()
```

The first line #!/usr/bin/env python enables the script to be executed directly from the terminal in Unix systems.

The text within """ is the docstring to document the script.

While python executes the code in the script line by line sequentially, it is a good practice to separate the definition of the functions and the program execution.

This is achieved by with the if __name__ == "__main__": condition at the end of the script, within which the main() function is executed.

In addition, this structure enables the functions defined in this script to be imported and executed by other scripts, enabling code reproducibility.

Docstrings

Docstrings are a block of code delimited by """ inserted right after the definition of a function, or at the start of a script, that easily document the functionality of that function, or script.

> ***Note:*** Using a standard structure for the docstring (Parameters, <argument> : <type>, Returns), online documentation becomes easier as several parsers exist for this documentation structure.

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2
```

```
    Returns
    -------
    bool
        Description of return value

    Examples
    --------
    >>> func(1, "a")
    True
    """

    return True
```

The docstring of a function is automatically interpreted by the interpreter as part of that function's documentation.

```
>>> print func.__doc__
Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    Examples
    --------
```

```
    >>> func(1, "a")
    True
>>>
```

Executable python scripts

Check here on how to run the python script without having to call the python interpreter through the command line.

Under Mac, Linux, BSD, Unix:

First line of the python script must include the path to the interpreter

script.py:

```
#!/usr/bin/env python

print 'This code was executed from a python script'
```

And the script file permissions must be set to executable

```
$ chmod +x script.py
$ ./script.py
This code was executed from a python script
$
```

Under Windows

Must call the python interpreter directly.

```
$ C:\Python27\python.exe
C:\Users\Username\Desktop\script.py
```

Practice exercises

Summary

This concludes the introduction to python. The python syntax and default data types were detailed. You are now able to use the python interpreter directly or run python scripts.