Python

What is python exactly?

> Python is a widely used high-level, general-purpose, interpreted, dynamic programming language.
> -- Wikipedia

So what does that mean? Let's start from the end.

Python is a **programming language**. This means python is a language with which we can communicate to our computer and tell it what to do!

A **dynamic** programming language, is a programming language that executes it's commands at runtime. On the other hand *static* programming languages require a procedure that is called *compilation*. This procedure first translates the human-readable commands into machine-language instructions. An **interpreted** language doesn't require a *compiler* to run, but an *interpreter*.

General-purpose programming languages, are those that are used in a wide variety of application domains. On the contrary there are *domain-specific* programming languages that are used in a single domain. Some examples of the latter are *HTML* (markup language for web applications), *MATLAB* (matrix representation and visualization), *SQL* (relational database queries) and *UNIX shell scripts* (data organization in unix-based systems).

A **high-level** programming language has a strong abstraction from the details of the computer. These languages are much easier to work with, because they automate many areas such as memory management.

Python is a really easy and powerful programming language and it has a simple and very straightforward syntax.

Versions

There are two python versions running in parallel, **python 2.7** and **python 3.x**. While python 3 came out in 2008, it saw a slow adoption from the community. We will be using python 3 for the remainder of this tutorial, but the code will be **fully compatible** with python 2.

Comments in python

Comments are lines that are ignored from the computer and are meant only for humans to be read.

In [14]:
```
# This is a comment!

'''
This is a
multiline
comment.
'''

"""
This is also a multiline comment.
' and " are interchangeable in python.
"""
```

Out[14]:
```
'\nThis is also a multiline comment.\n\' and " are interchangeable in pytho
n.\n'
```

Printing to the screen

The first thing we want to do when learning any programming language is print something (like *"Hello World"*) on screen.

In order to display information to the user, we use the `print` function. This also includes a *new line* directive (i.e two prints are shown in separate lines).

In [15]:
```
print('Hello World!')
```

```
Hello World!
```

This line instructs the computer to display the phrase 'Hello World!' on screen.

The `print` function is one of the major differences between python 2 and 3. In python 2.x the correct syntax would be:

```
print 'Hello World!'
```

We can also print multiple things at once:

In [16]:
```
print('argument 1, ', 'argument 2, ', 'argument 3')
```

```
argument 1,  argument 2,  argument 3
```

In order to get the same result for both versions of python, we could add the line:

In [17]:
```
from __future__ import print_function
print('argument 1, ', 'argument 2, ', 'argument 3')
```

```
argument 1,  argument 2,  argument 3
```

Importing external libraries

The first command is what we call an ***import***. These extend the functionality of python by adding more commands (or in this case modifying existing ones). This is done by *imorting* an **external library** (in this case the `__future__` library).

Libraries are a really important part of programming, because they allow us to use code that is already written by others. This way we don't have to write every program from scratch!

Let's say we want to create an *array*. Python does not support arrays, but luckily there is an external library that does: **numpy**.
We can use external libraries (like *numpy*) in three ways:

In [18]:
```
import numpy  # imports the library as is
numpy.array([1,2,3])  # creates the array [1,2,3]

import numpy as np  # imports numpy and from the future we can refer to it as 'np
np.array([1,2,3])  # creates the array [1,2,3]

from numpy import array  # imports only the class 'array' from the library numpy
array([1,2,3])  # creates the array [1,2,3]
```

Out[18]:
```
array([1, 2, 3])
```

We can choose any way we like to import libraries, each has it's pros and cons. Note that we only need to import a library **once** in our program to use it!

The main repository for python packages is PyPI (https://pypi.python.org/pypi), and the main tool for dowloading and installing packages from this repositpry is called pip (https://pip.pypa.io/en/stable/). In order to install pip, download get-pip.py (https://bootstrap.pypa.io/get-pip.py) and run the downloaded script with the following command:

```
python get-pip.py
```

Once *pip* is installed, to download and install a new package (e.g. *numpy*) just type:

```
pip install numpy
```

or

```
python -m pip install numpy
```

Assigning values to variables

To store information in memory we use *variables*. These can be letters or combinations of letters and numbers that help us store data to memory.

The procedure with which we give a variable a value is called **assignment** and is done in python with the **equal sign (=)**.

In [19]:
```
number_1 = 66
```

In the previous line we instructed the computer to store the number *66* in it's memory. We also told the computer, that from now on, we will refer to this number as *number_1*.

Now, if we want to print the stored number, we just need to reference it.

In [20]:
```
print(number_1)
```

```
66
```

Once a value has been assigned to a variable, it **can** be changed.

In [21]:
```
number_1 = 55
print(number_1)
```

```
55
```

Whitespace is good for making code easier to read by humans, but it makes **no** difference for the computer.

In [22]:
```
a=5
print(a)
a               =   5
print(a)
```

```
5
5
```

These two are exactly the same.

One exception would be **whitespace before the commands**. This is also called **indentation**. Indentation, as we'll see in the future, is important for python to understand nested commands!

In [23]:
```
print(a)
   print(a)
```

```
  File "<ipython-input-23-1ef56d9202ee>", line 2
    print(a)
    ^
IndentationError: unexpected indent
```

We are not allowed to indent commands without reason!

When python encounters a state it is not meant to, it usually raises an **error**! There are several built-in error types (such as the *IndentationError* we saw before). Furthermore, we can *create* our own errors and *handle* them accordingly, in order to prevent our program from doing things it is not meant to! We'll learn how to do this in a later tutorial.

The process of assigning a value to a variable binds the variable's name (e.g *number_1*) to that value. If we wish to free the name for future use we can **delete** the variable. This causes the stored memory to be lost!

In [24]:
```
del number_1
print(number_1)
```

```
---------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-24-dd2244dd4a29> in <module>()
      1 del number_1
----> 2 print(number_1)

NameError: name 'number_1' is not defined
```

We got this error because we tried to print the value of a variable that didn't exist (because we deleted it in the previous line).

Variable deletion should be done **only** when we **no longer need** the information this variable stores.

Similar to assignments, del can also delete multiple objects at once.

When naming variables keep in mind that python is **case sensitive**.

In [25]:
```
number_1 = 1
Number_1 = 2
NUMBER_1 = 3
nUmBeR_1 = 4
print(number_1, Number_1, NUMBER_1, nUmBeR_1)
```

```
1 2 3 4
```

In python these are 4 different variables.

A good way of presenting data to the user is adding a description and then the value.

In [26]:
```
print('The value stored in the variable nUmBeR_1 is:', nUmBeR_1)
```

```
The value stored in the variable nUmBeR_1 is: 4
```

In python we prefer `lower_case_variable_names_separated_with_underscores` .

Data Types

The data we store in a variable is always of a certain **type**. In other programming languages we would have to declare the type of data that the variable stores beforehand. However, in python the variable has it's type appointed dynamically when it is assigned a value.

Numberic

We'll see two main numeric data types, **integers** and **floats**

In [27]:
```
i = 13   # an integer
f = 8.2  # a float
```

Python supports multiple operations between numerical data types:

In [28]:
```
a = 5    # integer
b = 2.2  # float

print('a + b  =', a + b)
print('a - b  =', a - b)
print('a / b  =', a / b)  # this wouldn't be the same in python 2
print('a * b  =', a * b)
print('a ** b =', a ** b)  # a to the power of b
```

```
a + b  = 7.2
a - b  = 2.8
a / b  = 2.2727272727272725
a * b  = 11.0
a ** b = 34.493241536530384
```

Multiple operations can be done in one line:

In [29]:
```
a + b / a ** (a - b)
```

Out[29]:
```
5.024283242041717
```

The priorities are the same as in algebra.

strings

A string is a sequence of characters **enclosed in quotes** (either 'single' or "double").

In [30]:
```
st1 = '   a sTRiNg '
st2 = "Another strinG     "
```

Operations work a bit differently with strings than with numbers.

In [31]:
```
print(st1 + st2)  # concatenates the two strings
print(st1 * 3)    # repeats the string 3 times
```

```
   a sTRiNg Another strinG
   a sTRiNg    a sTRiNg    a sTRiNg
```

Strings have many built-in methods that could prove useful.

In [32]:
```
print(st1, '-->', st1.lower())  # converts the capital letters to lower-case
print(st1, '-->', st1.strip())  # remove whitespace from both left and right
print(st1, '-->', st1.replace('i', '1'))  # replace all 'i's with '1's (case-sens
```

```
   a sTRiNg  -->    a string
   a sTRiNg  --> a sTRiNg
   a sTRiNg  -->    a sTR1Ng
```

We can split a string to a *list* of substrings.

In [33]:
```
st3 = 'one third string'
print(st3, '-->', st3.split())  # split the string by considering the spaces
print(st3, '-->', st3.split('i'))  # split the string by considering the 'i's
```

```
one third string --> ['one', 'third', 'string']
one third string --> ['one th', 'rd str', 'ng']
```

One of the most useful operations involving strings is formatting. With this we can input the values of variables into strings.

In [34]:
```
print('var1 = {}, var2 = {}'.format(a, b))
```

```
var1 = 5, var2 = 2.2
```

lists

Lists are sequences that can store any object in python. Lists are the most versatile data structure in python and can be defined by square brackets `[]` .

In [35]:
```
ls1 = ['a', 'b', 'c', 1, 2, 3, 5.5]
print(ls1)
```

```
['a', 'b', 'c', 1, 2, 3, 5.5]
```

We can access a single element of a list through its **index**. Because a list is an ordered data structure, the index of each element is its position minus one (the index starts from 0).

In [36]:
```
print(ls1[0])   # first element
print(ls1[2])   # third element
print(ls1[-1])  # last element
```

```
a
c
5.5
```

Slicing is an operation with which we keep a **subset** of the original list.

In [37]:
```
print(ls1[1:6:2])  # slice from second (index: 1) to sixth with a step of 2
print(ls1[:3])     # slice of the first three elements
print(ls1[-2:])    # slice of the last two elements
```

```
['b', 1, 3]
['a', 'b', 'c']
[3, 5.5]
```

There are many ways we can manipulate the elements in a list, the most common are:

In [38]:
```
print('original list:', ls1)
ls1.insert(2, 'new element')
ls1.append('last')
del ls1[5]
ls1.remove('c')
print('final list:', ls1)
```

```
original list: ['a', 'b', 'c', 1, 2, 3, 5.5]
final list: ['a', 'b', 'new element', 1, 3, 5.5, 'last']
```

Python has many built-in functions that may prove useful when dealing with lists.

In [39]:
```
ls2 = [1, 5, 4, 7, 2, 8, 4, 2, 6]
print('list:   ', ls2)
print('max:    ', max(ls2))
print('min:    ', min(ls2))
print('sum:    ', sum(ls2))
print('length: ', len(ls2))
print('sorted: ', sorted(ls2))
```

```
list:    [1, 5, 4, 7, 2, 8, 4, 2, 6]
max:     8
min:     1
sum:     39
length:  9
sorted:  [1, 2, 2, 4, 4, 5, 6, 7, 8]
```

lists can also contain other lists as their elements

In [40]:
```
ls1[2] = ['inner1', 'inner2', 'inner3']
print(ls1)
```

```
['a', 'b', ['inner1', 'inner2', 'inner3'], 1, 3, 5.5, 'last']
```

To access an element in the inner list:

In [41]:
```
ls1[2][0]  # the first element of the 'inner' list, that is placed in the second
```

Out[41]:
```
'inner1'
```

tuples

Tuples are immutable lists. Once they have been created we can't make any changes. Tuples are defined inside parentheses `()`.

In [42]:
```
tup = (1, 2, 3)

# The following would raise errors:

# tup[1] = 4
# tup.append(6)
# del tup[0]
```

dictionaries

Dictionaries are data structures organized by **keys** and **values**. Each element has a **key**, with which we can access it. Dictionaries are created with curly brackets `{}`.

In [43]:
```
dict1 = {'k1': 1,
         'k2': 'abc',
         'k3': [1, 2, 3, 'a', 'b', 'c']}

dict1
```

Out[43]:
```
{'k1': 1, 'k2': 'abc', 'k3': [1, 2, 3, 'a', 'b', 'c']}
```

In [44]:
```
dict1['k2']
```

Out[44]:
```
'abc'
```

Dictionaries have no sense of order. To add/remove elements in an existing dictionary:

In [45]:
```
dict1['k4'] = 'new element'  # add a new element with 'k4' as its key
del dict1['k3']
dict1
```

Out[45]:
```
{'k1': 1, 'k2': 'abc', 'k4': 'new element'}
```

We can see all keys and values in a dictionary like this:

In [46]:
```
print(dict1.keys())
print(dict1.values())
```

```
dict_keys(['k1', 'k2', 'k4'])
dict_values([1, 'abc', 'new element'])
```

Logical operations

The main data type associated with logical operations is `bool`, which stands for *boolean* and can take two values: `True` and `False`.

In [47]:
```
a = 5
b = 3

print('{} == {}: {}'.format(a, b, a == b))  # Returns True if the two operands ar
print('{} != {}: {}'.format(a, b, a != b))  # Returns True if the two operands ar
print('{} > {}:  {}'.format(a, b, a > b))   # Returns True if the left operand is
print('{} < {}:  {}'.format(a, b, a < b))   # Returns True if the right operand i
print('{} >= {}: {}'.format(a, b, a >= b))  # Returns True if the left operand is
print('{} <= {}: {}'.format(a, b, a <= b))  # Returns True if the right operand i
```

```
5 == 3: False
5 != 3: True
5 > 3:  True
5 < 3:  False
5 >= 3: True
5 <= 3: False
```

There are three logical operators in python: `and`, `or`, and `not`.

In [48]:
```
t = True
f = False

print('{} and {}: {}'.format(t, f, t and f)) # Returns True if both operands are
print('{} or {}:  {}'.format(t, f, t or f))  # Returns True if one of the two ope
print('not {}:       {}'.format(t, not t))   # Returns the reverse logical state
```

```
True and False: False
True or False:  True
not True:       False
```

The main use of logical operations is with `if` statements. An `if` statement allows us to control the *flow* of the program by selecting which operations will be executed, depending on the outcome of the `if` statement.

For example

In [49]:
```
if a < b: # if the condition is true then execute the indented code below
    print('a is smaller than b') # note the indent
else: # if the condition is not true then execute the indented code below
    print('a is not smaller than b')
```

```
a is not smaller than b
```

The first `if` evaluates if the contidion is `True` or `False`. If it is `True` the code underneath is executed. If not then it skips to the operations underneath `else`.

We can check for more than one condition through `elif`:

In [50]:
```
if a < b:  # if the condition is true then execute the indented code below
    print('a is smaller than b')  # indented line
elif (a == b):  # if the first condition is not true but this one is then execute
    print('a is equal to b')
else:  # if none of the above conditions aren't true then execute the indented co
    print('a is larger than b')
```

```
a is larger than b
```

Iterations

Iterations can be used for running the same statements for a number of times. The most common way of iterating is through `for`.

In [51]:
```
for element in ls1:
    print(element)
```

```
a
b
['inner1', 'inner2', 'inner3']
1
3
5.5
last
```

`element` takes the values of `ls1` one at a time. A common way of using `for` loops is along with `range`. `range` is a special type of list that contains sequential integers.

In [52]:
```
for i in range(10):  # range 10 returns 10 integers from 0 to 9
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```