



Introduction to Python course – part I

Silvia Salatino, PhD

17.10.2019 – Wellcome Centre for Human Genetics, Oxford

What is Python?



Named after the British comedy group Monty Python, Python was released in 1991 by its creator, Guido van Rossum

(this guy)



“Python is an interpreted, high-level, general-purpose programming language.”

(source: Wiki)

Instructions are executed directly, without needing to previously compile them into machine code.

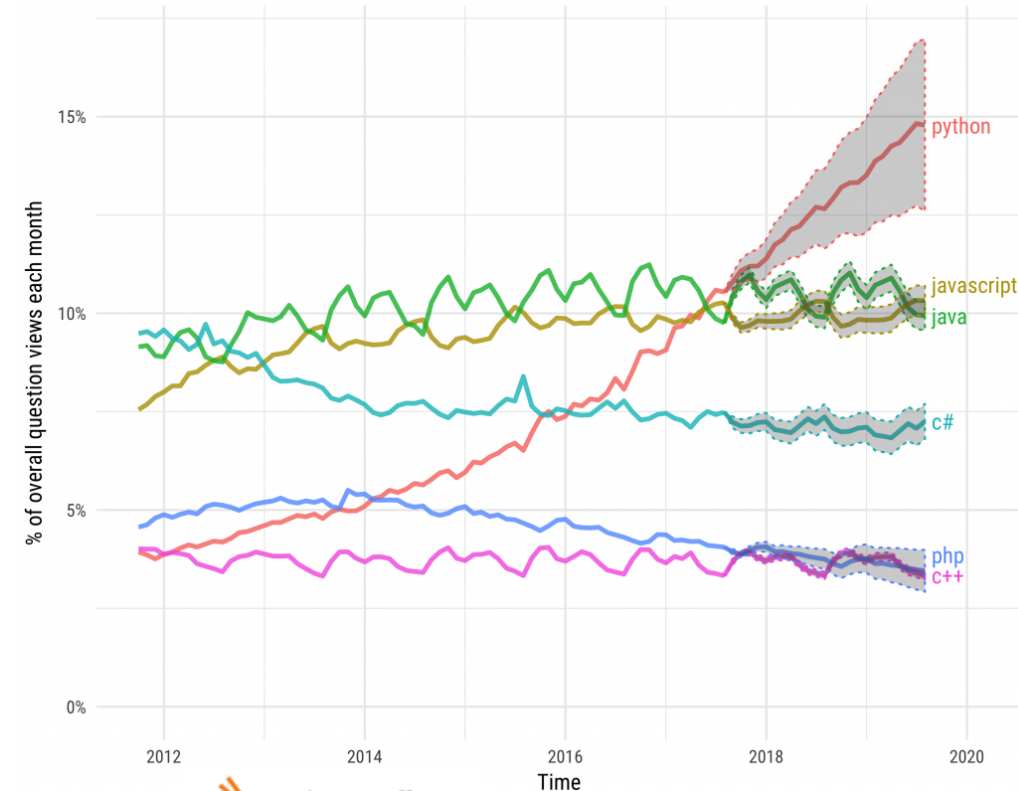
It is meant to be an easily readable language, using many English keywords and little punctuation

Can be used for writing software in a wide variety of application domains (as opposed to, e.g., page description or database query languages).

...and why is it so popular?

Projections of future traffic for major programming languages

Future traffic is predicted with an STL model, along with an 80% prediction interval.



source: stackoverflow

The Python community is very active, with several conferences and workshops taking place all around the globe, and a well-documented webpage (www.python.org):

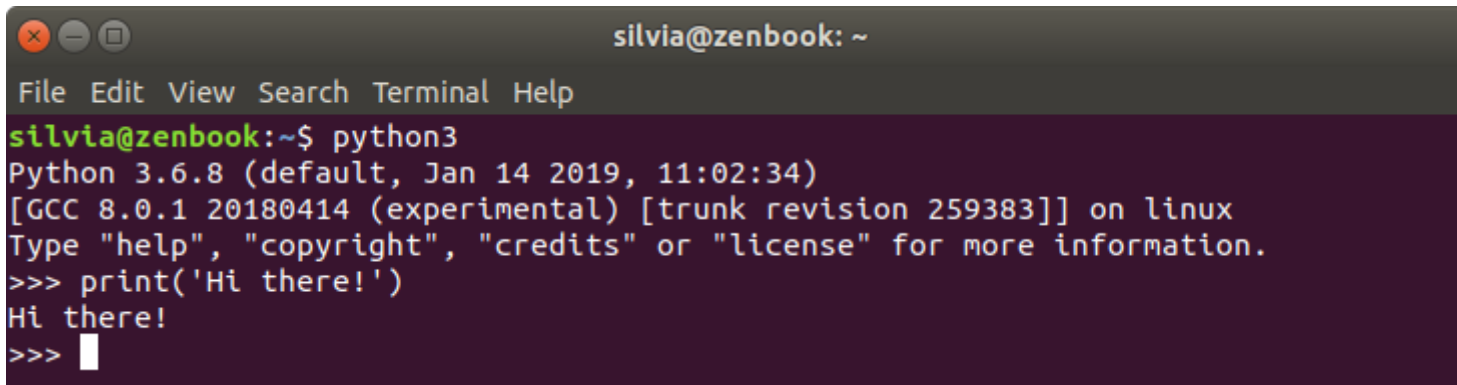
Thanks to its many plugins and third-party libraries, Python is widely used (and getting even more popular) in **scientific computing**, **data science** and **machine learning**.



Okay, but... how do I use Python in practice?

There are several ways to use the *Python interpreter*:

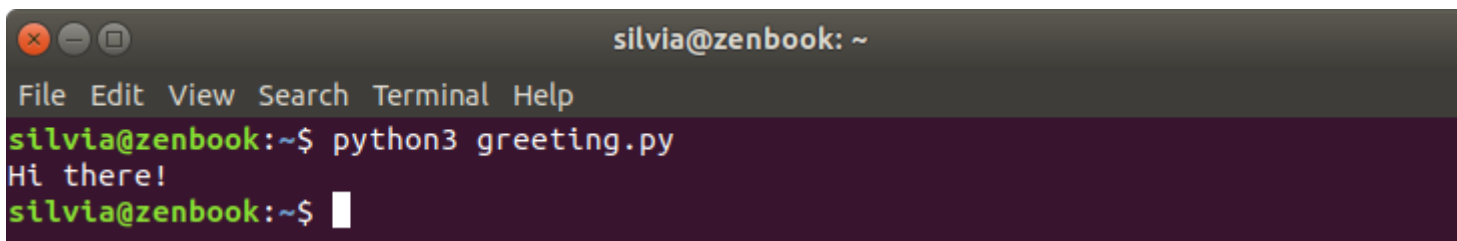
- **Interactive mode**: open a terminal and type either `python` or `python3` (depending on how it was installed). This will start the *Python prompt* or *shell* (usually indicated by `>>>`), in which you can type individual commands:

A terminal window titled 'silvia@zenbook: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'silvia@zenbook:~\$'. The user has entered 'python3', which has started the Python 3.6.8 interpreter. The interpreter displays version and build information, then the prompt '>>>'. The user has entered 'print('Hi there!')', and the interpreter has printed 'Hi there!'. The prompt '>>>' is shown again with a cursor.

```
silvia@zenbook:~$ python3
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hi there!')
Hi there!
>>>
```

Once you've finished, press CTRL+D or type `quit()` to exit the interpreter.

- **Script mode**: with any text editor, write your code in a file (by convention, ending with the suffix `.py`) and then execute this *script* from command-line:

A terminal window titled 'silvia@zenbook: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'silvia@zenbook:~\$'. The user has entered 'python3 greeting.py', and the interpreter has printed 'Hi there!'. The prompt 'silvia@zenbook:~\$' is shown again with a cursor.

```
silvia@zenbook:~$ python3 greeting.py
Hi there!
silvia@zenbook:~$
```

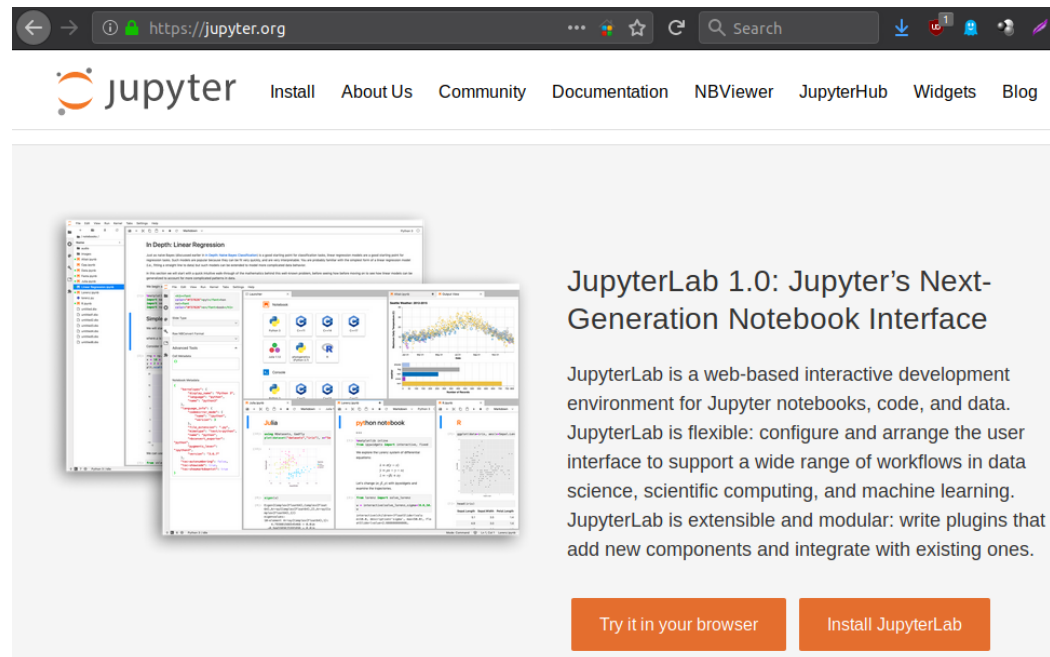
Okay, but... how do I use Python in practice?

- Integrated Development Environment (IDE):

An IDE is an environment which provides many features (like coding, debugging, executing, auto-complete, auto-linting, manage files and libraries, version control, etc.) in one place, making programmers' tasks much easier.

IDEs have an in-build terminal and one or more **kernels** (i.e. computational engines to execute the code).

There are several IDEs that work with Python (e.g. **Atom**, **Sublime**, **Eclipse**), some of which specifically made for Python (e.g. **PyCharm**, **Spyder**, **PyDev**). In this course we will use **JupyterLab**.

The image shows a screenshot of the JupyterLab 1.0 website. At the top, there's a navigation bar with the Jupyter logo and links for Install, About Us, Community, Documentation, NBViewer, JupyterHub, Widgets, and Blog. Below the navigation bar, there's a large section titled "JupyterLab 1.0: Jupyter's Next-Generation Notebook Interface". To the left of this text is a collage of images showing the JupyterLab interface in use, including code editors, notebooks, and data visualizations. To the right of the collage, there's a paragraph describing JupyterLab as a web-based interactive development environment for Jupyter notebooks, code, and data. It mentions that JupyterLab is flexible and extensible. At the bottom of this section, there are two orange buttons: "Try it in your browser" and "Install JupyterLab".

https://jupyter.org

jupyter Install About Us Community Documentation NBViewer JupyterHub Widgets Blog

JupyterLab 1.0: Jupyter's Next-Generation Notebook Interface

JupyterLab is a web-based interactive development environment for Jupyter notebooks, code, and data. JupyterLab is flexible: configure and arrange the user interface to support a wide range of workflows in data science, scientific computing, and machine learning. JupyterLab is extensible and modular: write plugins that add new components and integrate with existing ones.

Try it in your browser Install JupyterLab

Introduction to IPython

IPython (= Interactive Python) is an alternative Python interpreter to the simple Python shell, with a number of useful advantages:

- support for native shell commands like `cd`, `ls`
- input and output history
- proper indentation and tab completion
- syntax highlighting
- documentation (type `?` after a given command, or type `%quickref` for a quick list of available commands)



```
IPython: home/silvia
File Edit View Search Terminal Help
silvia@zenbook:~$ ./anaconda3/bin/ipython
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import math

In [2]: math.log10(100)
Out[2]: 2.0

In [3]: math.sin(3)
Out[3]: 0.1411200080598672

In [4]: In
Out[4]: ['', 'import math', 'math.log10(100)', 'math.sin(3)', 'In']

In [5]: Out
Out[5]:
{2: 2.0,
 3: 0.1411200080598672,
 4: ['', 'import math', 'math.log10(100)', 'math.sin(3)', 'In', 'Out']}

In [6]:
```

```
In [4]: map?
Init signature: map(self, /, *args, **kwargs)
Docstring:
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables. Stops when the shortest iterable is exhausted.
Type:          type
Subclasses:
```



“**In**” is a list, keeping track of commands in order
“**Out**” is a dictionary, mapping commands to their outputs

The so called “*magic commands*”

The ***magic commands***, prefixed by the `%` character when referred to a line or by `%%` when referred to a cell, are a powerful advantage of IPython over Python native shell.

A full list can be displayed with `%lsmagic` and some examples include:

- `%cd` and `%pwd` → to change and display the current working directory
- `%matplotlib inline` → to set up matplotlib to work interactively and display plots
- `%time` and `%timeit` → to measure the execution time of a Python statement or expression
- `%mkdir` → to create a new folder without leaving the IPython session
- `%run` → to execute external commands
- etc...

Moreover, any command following the exclamation mark `!` on a given line will be executed not by the Python kernel, but by the system command-line, thus enabling you to run non-Python code from the same terminal, without having to switch between different terminals.

Setup

JupyterLab is a browser-based graphical interface to the IPython shell and, in addition, it allows the user to include formatted text (e.g. markdown), plots, mathematical equations, and much more.

Open a browser and go to the URL <https://jupyter.org/try>. Click on “Try JupyterLab” and wait for being redirected to a new page. Close the “Lorenz.ipynb” example tab, but keep the “Reference” tab open, which might turn out useful.

Click on “+” to open the Launcher and then select “Python 3” under the “Notebook” section.

At the end of this session, you can rename and export your Notebook in HTML or other formats.

The image is a composite showing the Jupyter website and a JupyterLab interface. The top part shows the Jupyter.org website with a 'Try JupyterLab' button highlighted by a large white arrow. The bottom part shows a screenshot of the JupyterLab interface with a file browser on the left, a code editor in the center, and a documentation panel on the right. A white arrow points to the '+' button in the file browser.

JupyterLab

Install About Us Community Documentation NBViewer JupyterHub Widgets Blog

Try Jupyter

You can try Jupyter out right now, without installing anything. Select an example below and you will get a temporary Jupyter server just for you, running on mybinder.org. If you like it, you can [install Jupyter](#) yourself.

Try Classic Notebook

A tutorial introducing basic features of Jupyter notebooks and the IPython kernel using the classic Jupyter Notebook interface.

Try JupyterLab

JupyterLab is the new interface for Jupyter notebooks and is ready for general use. Give it a try!

Try Jupyter with Julia

A basic example of using Jupyter with Julia.

Try Jupyter with R

Try Jupyter with C++

Try Jupyter with Scheme

JupyterLab

[Docs](#) » JupyterLab Documentation [Jupyter](#) | [Edit on GitHub](#)

JupyterLab Documentation

JupyterLab is the next-generation web-based user interface for Project Jupyter. [Try it on Binder](#). JupyterLab follows the Jupyter [Community Guides](#).

The screenshot shows the JupyterLab interface. On the left is a file browser with a table of files and folders. In the center is a code editor with a Python notebook. On the right is a documentation panel for JupyterLab. A white arrow points to the '+' button in the file browser.

| Name | Last Modified |
|-----------------------|---------------|
| data | 3 months ago |
| notebooks | 3 months ago |
| TCGA_Data | 3 months ago |
| big.csv | 3 months ago |
| jupyterlab-slides.pdf | 3 months ago |
| jupyterlab.md | 3 months ago |
| Lorenz.ipynb | 3 months ago |
| lorenz.py | 3 months ago |
| markdown_python.md | 3 months ago |

The Lorenz Differential Equations

Before we start, we import some preliminary libraries. We will also import (below) the accompanying `lorenz.py` file, which contains the actual solver and plotting routine.

```
[ ]: %matplotlib inline
from ipywidgets import interactive, fixed
```

We explore the Lorenz system of differential equations:

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Let's change (σ, β, ρ) with `ipywidgets` and examine the trajectories.

```
[ ]: from lorenz import solve_lorenz
w=interactive(solve_lorenz,sigma=(0.0,50.0),rho=(0.0,50.0),
w
```

For the default set of parameters, we see the trajectories swirling around two points, called attractors.

The object returned by `interactive` is a `Widget` object and it has attributes that contain the current result and arguments:

```
[ ]: t, x, t = w.result
```

Getting Started

Introduction to JupyterLab

Notebooks are made up of one or more **cells**. Each cell can contain either Python code or Markdown. You can switch between these using the dropdown menu “**Code**”.

For executing code, it relies on a running Python process, called **kernel**. In this case, the kernel we will use today is Python 3.

To execute a cell click on it and press **SHIFT+ENTER** on your keyboard. The output of the cell will be displayed beneath the cell. Cells are executed from top to bottom and failing to run cells in order can result in errors.

The next slides will focus on Python data structures – the theory part is written as comments (**#**) on the right of the commands.

JupyterLab - Mozilla Firefox

JupyterLab

https://hub.gke.mybinder.org/user/jupyterlab

File Edit View Run Kernel Tabs Settings Help

Untitled.ipynb Python 3

Code

```
[1]: x = 1 + 2
     print(x)
3
```

```
[2]: y = 7 + x
     print(y)
10
```

This cell contains **Markdown** (a useful `_markup` language to write ``code`` and rich text)

This cell contains **Markdown** too, and has just been executed with **SHIFT+ENTER**

```
[3]: %matplotlib inline
     import matplotlib.pyplot as plt
     plt.plot(x, y, 'bo')
```

```
[3]: [<matplotlib.lines.Line2D at 0x7fd61d462160>]
```

0 \$ 2 Python 3 | Idle Mode: Edit Ln 1, Col 1 Untitled.ipynb

Using Python as a calculator – integers and floats

Integer numbers (e.g. 1, 3, 26, etc.) have type `int`, whereas numbers with a fractional part (e.g. 2.4, 8.0) have type `float` (which stands for “floating point number”).

```
[1]: 1 + 2 #the result will be an integer
[1]: 3

[2]: 3 / 4 #the result will be a floating point number
[2]: 0.75

[3]: 10 / 2 #division always returns a float (from Python 3 onwards)
[3]: 5.0

[4]: type(10 / 2) #the command type returns the type of an object
[4]: float

[5]: 2 ** 7 #this means 2 to the power of 7
[5]: 128

[6]: 17 / 3
[6]: 5.666666666666667

[7]: 17 // 3 #this operator returns the floor division
[7]: 5

[8]: age = 2019 - 1986 #the equal sign assigns a value to a variable
[9]: age #no result is displayed until the next interactive prompt
[9]: 33

[10]: city #if a variable is not defined (i.e. assigned a value) it will result in an error

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-aab2fe13cf24> in <module>
----> 1 city #if a variable is not defined (i.e. assigned a value) it will result in an error

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

Using Python to parse characters – strings (1)

Python can also manipulate strings, `str`, which are enclosed in either **single or double quotes**.

```
[11]: 'this is a string' #strings can be enclosed in single quotes
[11]: 'this is a string'
[12]: "and this one too!" #...and also in double quotes
[12]: 'and this one too!'
[13]: 'I don\'t mind quotes' #single quotes can be escaped with a backslash
[13]: "I don't mind quotes"
[14]: "I don't mind quotes" #...or with double quotes
[14]: "I don't mind quotes"
[15]: print("I don't mind strings") #print produces a more readable output
      I don't mind strings
[16]: print('nor\nnewlines') #...and interpretes escaped special characters properly
      nor
      newlines
[17]: 'My name is ' + 'Silvia' #strings can be concatenated with '+'
[17]: 'My name is Silvia'
[18]: 'I am ' + str(33) + 'years old' #to concatenate strings and numbers, these must be converted to strings
[18]: 'I am 33years old'
[19]: word = 'orange' #strings can be indexed
      word[2] #using a 0-based counter from the left-hand side
[19]: 'a'
[20]: word[-1] #or from the left-hand side for negative indexes
[20]: 'e'
[21]: word[1:4] #strings can be sliced from position i (inclusive) to i+n (exclusive)
[21]: 'ran'
```

Using Python to parse characters – strings (2)

```
[22]: word[:2] #omitting the first index, you can slice 'from the beginning'
```

```
[22]: 'or'
```

```
[23]: word[4:] #omitting the second index, you can slice 'until the end'
```

```
[23]: 'ge'
```

```
[24]: word[0] = 'g' #strings are immutable, you cannot assign a new value to a position
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-24-50399318b167> in <module>  
----> 1 word[0] = 'g' #strings are immutable, you cannot assign a new value to a position  
TypeError: 'str' object does not support item assignment
```

```
[25]: 'g' + word[1:] #...but if you need a different string, you can create it
```

```
[25]: 'grange'
```

```
[26]: len(word) #len is used to check the length of a given string
```

```
[26]: 6
```

Combining many values – lists

To group together multiple values, Python has **compound data types**. An example is the list, a comma-separated collection of values which is initialised either using **square brackets** or the `list()` function. Differently from other compound data types which we won't cover in this course (like **tuples**), lists are more versatile as they are mutable.

A list can be sorted in-place using the method `sort()`, or the not-in-place function `sorted()`.

```
[27]: fruits = ['orange', 'apple', 'kiwi', 'banana']
      fruits[2] #like strings, lists can be indexed, returning the item

[27]: 'kiwi'

[28]: fruits[2:] #...and like strings, they can be sliced, returning a new list

[28]: ['kiwi', 'banana']

[29]: fruits[0] = 'lime' #but unlike strings, they are mutable
      fruits

[29]: ['lime', 'apple', 'kiwi', 'banana']

[30]: fruits + [3, 1.24] #lists support concatenation and different data types

[30]: ['lime', 'apple', 'kiwi', 'banana', 3, 1.24]

[31]: fruits #...but this way the list is unchanged!

[31]: ['lime', 'apple', 'kiwi', 'banana']

[32]: fruits.append(3) #to change it, use append
      fruits.append(1.24)
      fruits

[32]: ['lime', 'apple', 'kiwi', 'banana', 3, 1.24]

[33]: fruits.remove('apple') #elements can also be removed
      fruits

[33]: ['lime', 'kiwi', 'banana', 3, 1.24]

[34]: fruits[:] = list() #...and replaced with an empty list ('[]' would have worked too)
      fruits

[34]: []

[35]: fruits = ['apple', 'kiwi'] #lists can also be nested
      vegetables = ['potato', 'spinach']
      f_and_v = [fruits, vegetables]
      f_and_v

[35]: [['apple', 'kiwi'], ['potato', 'spinach']]

[36]: len(fruits) #the len function works for lists too

[36]: 2
```

Manipulating groups of unsorted unique elements – sets

Sets are unordered collection of non-duplicated elements and support mathematical operations. They can be initialised either using **curly brackets** or the `set()` function.

```
[37]: fruits = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}  
fruits
```

```
[37]: {'apple', 'banana', 'orange', 'pear'}
```

```
[38]: 'orange' in fruits  #membership testing
```

```
[38]: True
```

```
[39]: 'basil' in fruits
```

```
[39]: False
```

```
[43]: x = set('oxford')  
y = set('cambridge')
```

```
[44]: x  #unique letters in x
```

```
[44]: {'d', 'f', 'o', 'r', 'x'}
```

```
[45]: y  #unique letters in y
```

```
[45]: {'a', 'b', 'c', 'd', 'e', 'g', 'i', 'm', 'r'}
```

```
[46]: x - y  #letters in x but not in y (difference)
```

```
[46]: {'f', 'o', 'x'}
```

```
[47]: x | y  #letters in x or y (union)
```

```
[47]: {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'i', 'm', 'o', 'r', 'x'}
```

```
[48]: x & y  #letters in x or y, or both (union)
```

```
[48]: {'d', 'r'}
```

```
[49]: x ^ y  #letters in x or y, but not in both (symmetric difference)
```

```
[49]: {'a', 'b', 'c', 'e', 'f', 'g', 'i', 'm', 'o', 'x'}
```

Manipulating groups of unsorted unique elements – dictionaries

Dictionaries are compound data structures indexed by keys, which are immutable and unique, and each **key** has a corresponding **value**, which could be a number, string, list, or even another dictionary.

Dictionaries can be initialised either using curly brackets, or by using the function `dict()`.

```
[50]: wheels = {'car': 4, 'motorcycle': 2, 'tricycle': 3}
wheels
```

```
[50]: {'car': 4, 'motorcycle': 2, 'tricycle': 3}
```

```
[51]: wheels['bus'] = 6
wheels
```

```
[51]: {'car': 4, 'motorcycle': 2, 'tricycle': 3, 'bus': 6}
```

```
[52]: del wheels['motorcycle'] #removing a key:value pair
wheels
```

```
[52]: {'car': 4, 'tricycle': 3, 'bus': 6}
```

```
[53]: wheels['bus'] #extracting the value corresponding to a given key
```

```
[53]: 6
```

```
[54]: list(wheels) #extracting all the keys
```

```
[54]: ['car', 'tricycle', 'bus']
```

```
[55]: 'bus' in wheels #checking for presence/absence of a key
```

```
[55]: True
```

```
[56]: 'airplane' in wheels
```

```
[56]: False
```


Practical session 

Practical session 1 – exercises

- 1) Using the string **s1 = 'CCGATTC'**, calculate its length, extract the subsequence **'ATT'** and use it to create a new string **s2** with suffix **'GG'**
- 2) Sort the list **chroms = ['chr13', 'chr1', 'chrX', 'chr8']**, remove **'chrX'** from it, append **'chrY'** and print the resulting list
- 3) Using the two sets **A = set([4, 7, 9, 2])** and **B = set([1, 2, 3, 4])**, print their union and symmetric difference
- 4) Using the dictionary **d = {'hs': 'Homo sapiens', 'mm': 'Mus musculus', 'bt': 'Bos taurus'}**, remove the cow, verify that it is no longer a key of this dictionary and add a new organism with **'rn'** as key and **'Rattus norvegicus'** as value

Practical session 1 – solutions

1) `len(s1) ; s2 = s1[3:6] + 'GG'`

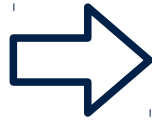
2) `chroms.sort() ; chroms.remove('chrX') ; chroms.append('chrY') ; print(chroms)`

3) `A | B ; A ^ B`

4) `del d['bt'] ; 'bt' in d ; d['rn'] = 'Rattus norvegicus'`

Flow control statements – “for” loop

Let's say you have a sequence of elements and you want to apply the same operation to each of them:



```
[1]: numbers = [10, 20, 30]
```

```
[2]: numbers[0] + 5
```

```
[2]: 15
```

```
[3]: numbers[1] + 5
```

```
[3]: 25
```

```
[4]: numbers[2] + 5
```

```
[4]: 35
```

The **flow control statements** in Python are very useful whenever you have to repeat the same operation for each element of a sequence (either a list or a string).



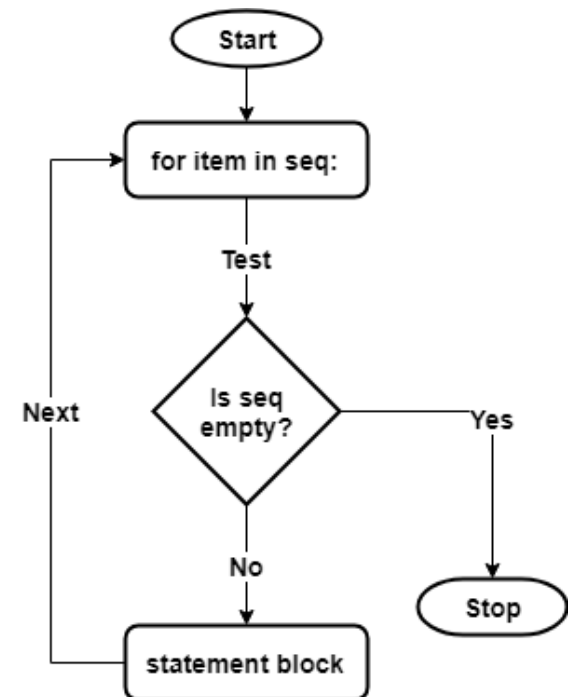
```
[5]: for n in numbers:  
      print(n + 5)  #add 5 to each element and print the sum
```

```
15
```

```
25
```

```
35
```

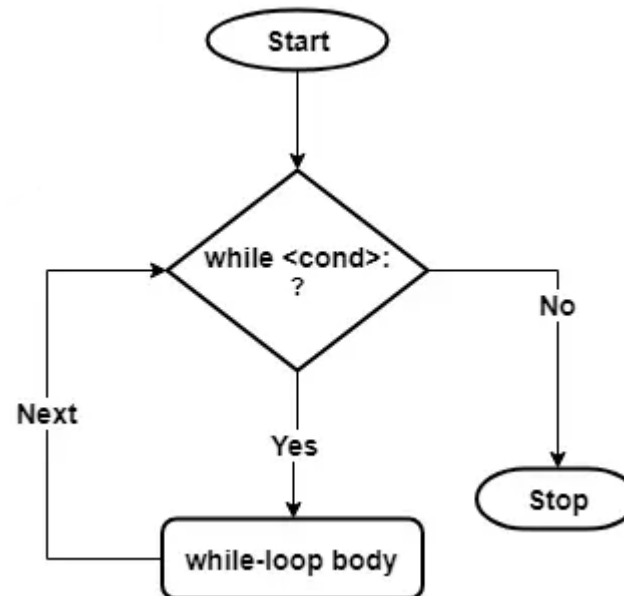
One type of flow control statements is the **“for” loop**, which iterates over the items of a sequence in the same order in which they appear in that sequence. The **body** of the loop needs to be **indented** with a tab or space(s) and Python will raise an error if the indentation is wrong!



Flow control statements – “while” loop

The for loop executes an operation until the last element of a sequence is reached.

Another flow control statement, the **“while” loop**, executes an operation on the elements of a sequence only if a given condition is met.



```
[3]: a = 1
while a < 10:
    a = a + 2
    print(a)
```

```
3
5
7
9
11
```

More control flow tools – if, elif, else, and, or, not

Python supports the following logical mathematical conditions:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

...which can be used in for and while loops to specify what to do (or not to do) when a given condition is met (or unmet).

The keywords to define and connect logical conditional statements are:

- **if** (one, required)
- **elif** (one or more, optional) short for “else if”
- **else** (optional)
- **or**
- **and**
- **not**

```
[1]: n = 3
      if n < 0:
          print(n, 'is negative')
      elif n == 0:
          print(n, 'is zero')
      else:
          print(n, 'is positive')
      print('This is always printed')
```

```
3 is positive
This is always printed
```

```
[2]: n = 3
      if n > 0 and n < 5:
          print('Both conditions are True')
```

```
Both conditions are True
```

```
[3]: a = 5
      b = 2
      c = 7
      if a > b or b > c:
          print('At least one of the conditions is True')
```

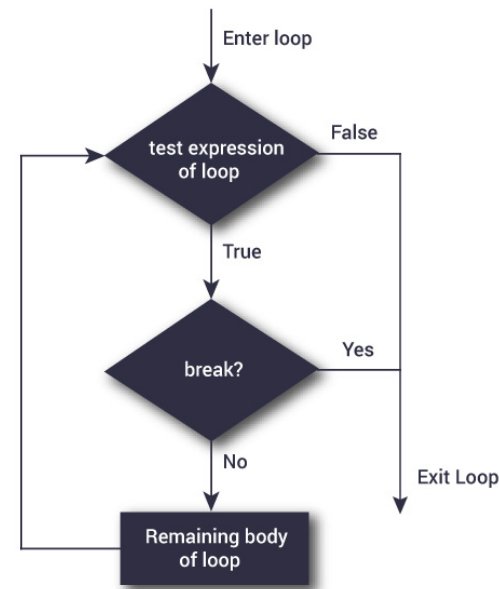
```
At least one of the conditions is True
```

```
[4]: a = 2
      if not a == 5:
          print('This is a negative statement')
```

```
This is a negative statement
```

More control flow tools – break, continue, pass

The **break** statement can alter the flow of a normal loop by stopping it



Source: www.programiz.com

```
[1]: a = 5
for b in range(100):
    if b > a:
        print(b)
        break
```

6

The **continue** statement skips to the next iteration of the loop

```
[2]: for letter in "example":
    if letter == "p":
        continue
    print(letter)
```

e
x
a
m
l
e

The **pass** statement is a placeholder for functionality to be added later and does not alter the flow.

```
[3]: new_list = []
for n in range(10):
    if n % 2 == 0:
        pass
    else:
        new_list.append(n)
print(new_list)
```

[1, 3, 5, 7, 9]

Practical session 🇨🇪

Practical session 2 – exercises

- 1) Using the dictionary **bases** = {'A': 'T', 'C': 'G', 'T': 'A', 'G': 'C'} and the sequence **orig_seq** = 'ATGGTCA', create a new sequence '**compl_seq**' with the complementary bases. Tip: you will need to use a **for** loop.
- 2) Using the **bases** dictionary from the previous exercise, skip every 'G' in the sequence **orig_seq** and exit the loop when you find a 'C'

Practical session 2 – solutions

```
1) bases = {'A': 'T', 'C': 'G', 'T': 'A', 'G': 'C'}
orig_seq = 'ATGGTCA'
compl_seq = ""
for letter in orig_seq:
    compl_seq += bases[letter]

print(compl_seq) # the result must be 'TACCAGT'
```

```
2) bases = {'A': 'T', 'C': 'G', 'T': 'A', 'G': 'C'}
orig_seq = 'ATGGTCA'
compl_seq = ""
for letter in orig_seq:
    if letter == 'G':
        continue
    elif letter == 'C':
        break
    else:
        compl_seq += bases[letter]

print(compl_seq) # the result must be 'TAA'
```

Thank you for your attention!

Questions?

silvia@well.ox.ac.uk

bioinformatics@well.ox.ac.uk