Python

What Python is?

- Python is a **easy to learn**, powerful programming language.
- It provides language constructs **as simple as sophisticated**, depending on the kind of user is programming with it
- It provides **efficient built-in data structures** and a simple approach to OOP.
  - It is not such efficient like C but it is extremely easier to use, it is not such structured like Java but it is more flexible.
- It is "**interpreted**"… like bash scripts, and it is **portable**.
  - But it can be compiled, too.
  - It is influenced by Modula-3, C. C++, Perl, Java, Unix shell and others.
- It has been **evolved during time**. First version on January 1994, then Python2 released on October 2000 and Python3 released on December 2008.
  - **python3 is backward-incompatible!!!** It means that syntax and standard libraries are not the same, but they are both large.
  - We will use Python3, https://docs.python.org/3.5
  - `print something` → `python2` `print(something)` → `python3`
- It is interactive: one can write and test programs directly form a terminal windows.
  - Extended tools there exists, such as IPython and Jupiter Notebook (jupyter.org)

Python

How to install it (under Ubuntu)?

```
sudo apt-get install python3
```

But let's have a better interface

```
sudo apt-get install ipython3
```

How to run it ?

`python3`

An interactive basic shell will answer you

```
~$python3
>>>print("Hello world")
Hello world
```

`ipython3`

An better interactive shell will answer you

`python3 myscript.py`

If you do not want an interactive instance, but you just want to execute a pre-written script.

How to install additional packages ? (python3 → pip3, python2 → pip)

```
sudo apt-get install pip3
sudo pip3 package_name
```

Python: built-in data types

Reading from standard input (console)

```
>>>a = input('please input the value of a')
5
>>>print(a)
5
```

We can also evaluate expressions

```
>>>a = eval(input('please input the value of a'))
5 + 5
>>>print(a)
10
```

Essential built-in data types

`int(a [,base])` for integers, ex. `10`

`float(a)` for floating point numbers, ex. `1.123`

`chr(a)` single characters, ex. `'a'`

`str(a)` strings, ex. `'this is a string'` or `"this is another string"` or

```
"""this is a multiple
lines string"""
```

`bool(a)` boolean values. It can be `True` or `False`. Everything is `0` or `None` is False.

`complex(a [,image])` for complex numbers

Python: operators

Numeric operators

`+`, `-`, `*`, `/` respectively addiction, subtraction, multiplication, division
`%` modulus (divides left hand operand by right and operand and returns the remainder)
`**` exponent (performs exponential (power) calculation
`//` floor division. The result is the quotient in which decimal digits are removed.

Bitwise operators

`&`, `|`, `^`, `~`, `<<`, `>>` respectively AND, OR, XOR, binary one complement, left shift and right shift

Assignment operators

`=`, `+=`, `-=` `*=`, `/=`, `%=`, `**=`, `//=`, `&=`, `|=`, `~=`, …
`++` and `--` are NOT in the python language

Python: operators

Comparison operators

`==` True if the values of the two operands are equal
`!=` True if the values of the two operands are not equal
`>, >, >=, <=` order comparisons

Logical operators

`and, or, not` are used for combining boolean (logical) expressions. For example

```
>>> a, b, c = True, True, False # NOTE this is a multiple assignment
>>> print( (a and b) or not c )
True
```

Identity operators

Python is almost like Java. Primitive data types (int, float, chr, bool, str, complex) are passed to functions by value, while other variables are passed by reference. This also means that structured objects are stored and passed by reference… we will discuss this later.

`is, is not` compare the memory locations of two objects. They compare the `id(x)` of the two operands, namely the pointer to them.

Python: operators

String operators

In python strings are immutable objects., thus they can not be modified.

`+` concatenation

```
>>>'hello' + 'world'
'hello world'
```

`*` repetition

```
>>>'AG' * 3
'AGAGAG'
```

`len(str)` length of a string

```
>>>len('AGAGAG')
6
```

Python: strings

String slice operators

For retrieving substrings or subsequences, but not only.

`s[index]` retrieves a single position/character from a string

```
>>>s='my string'
>>>s[0]
'm'
```

`s[start:end]` retrieves a substring, from the start position to the position before end

```
>>>s[0:5]
'my st'
```

`s[start:end:step]` retrieves a subsequence from start to end by jumping step position at time

```
>>>s='0123456789'
>>>s[0:len(s):3]
'0369'
```

Python: strings

String slice operators

Some useful examples.

Positions ca be negative indexes. For example, for retrieving the last element

```
>>>s='0123456789'
>>>s[-1]   # equivalent to s[-1:] or s[len(s) - 1:len(s)]
'9'
```

But we can also retrieve the entire string without the last element

```
>>>s[:-1]
'012345678'
```

We can also do this

```
>>>s[::]
'0123456789'
```

But more important, we can do this other!

```
>>>s[::-1]
'98765433210'
```

>>>What's that?

The reverse string!!! by reading the whole string with a step equal to -1

Python: strings

Built-in string functions

`str.find(sub[, start, end])` return the lowest position in the string where substring sub is found (in the slice `[start:end]`).
`str.index(sub[,start, end])` like `find()`, but raise `ValueError` when the substring is not found.

`str.replace(old, new [,count])` return a copy of the string (remember that strings are immutable), with all the (first `count`) occurrences of `old` are replaced with `new`.

`str.count(sub[, start, end])` return the number of occurrences of `sub` found in the string (in the slice `[start:end]`)

`str.upper(), str.lower()` return a copy of the string with all the characters converted to [upper/lower]case.

`str.isupper(), str.islower()` test for [upper/lower]case
`str.isalpha(),str.isdigit(), str.isprintable(), ...`

`str.strip([chars])` return a copy of the string with the leading and trailing characters in the list `chars` are remove. If the argument is omitted or equal to None, then whitespaces are removed.
`str.startswith(sub), str.endswith(sub)` return True if the string starts/ends with the specified substring.

Python: control flow statements

The conditional statement

```
if <condition>:
    <code block>
elif <condition>:
    <code block>
else:
    <code block>
```

In python a code block is not delimited by parentheses, instead indentation provides it

The while statement

```
while <condition>:
    <code block>
else:
    <code block>
```

The `else` statement is executed as soon as the `while` condition becomes `False`. This behavior is not maintained if a `break` instruction is used to exit the loop.

```
>>>a,i = 3,0
>>>while i < a :
>>>    print(i,' ', sep='',end='')
>>>    i += 1
>>>else:
>>>    print('finish')
0 1 2 finish
```

Python: control flow statements

The for statement

```
for object in list:
    <code block>
else:
    <code block>
```

A practical example is given by

```
>>>for i in range(3):
>>>    print(i,'', end='')
0 1 2
```

The **range** function provides a way to generate lists. Actually, it generates an **iterator** to a **virtual list**, for example range(0,100000000) does not generate a list of 100000000 elements, but it gives an "object" that iterates from 0 to 100000000 – 1

`range(stop)` it implicitly starts from 0

`range(start, stop[, step])` iterate from start (included) to end (not included) with a given step

```
s='my string'
start = 0
end = len(s)
step = 1
```

```
i = start
while i < end:
    print(s[i])
    i += step
```

```
for i in range(start,end,step):
    print(s[i])
```

These two loops are equivalent

Python: built-in data structures

Which are the python3 built-in data structures?

The standard python library provides a set of built-in data structures. The most important are **lists, tuples, sets, forzensets**, and **dictionaries**.

Lists and **tuples** are ordered sequences of objects. Unlike strings that contain only characters, list and tuples can contain any type of objects. Lists and tuples are like arrays. Tuples like strings are immutable. Lists are mutable so they can be extended or reduced at will.

Sets are mutable unordered sequence of unique elements whereas **frozensets** are immutable sets.

Dictionaries are mutable mappings of a list of keys to a corresponding list of values.

Python: lists

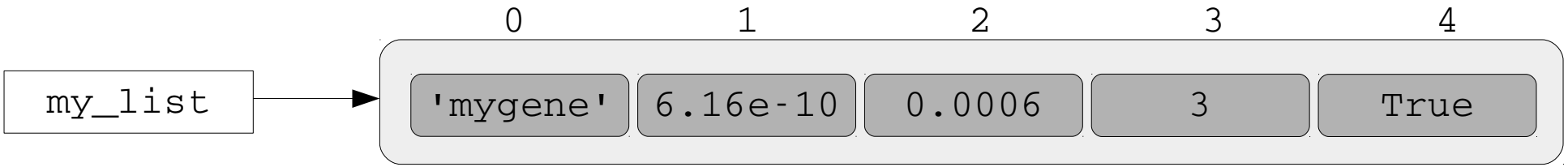
Lists

A list is a **mutable ordered** set of values, possibly of different types

```
['mygene', 6.16e-10, 0.0006, 3, True]
```

A variable can be created to hold a list

```
my_list = ['mygene', 6.16e-10, 0.0006, 3, True]
```



That variable is simply a **pointer/reference** to the real list.
It can be used to access the list, i.e. for reading list elements

```
>>>print(my_list[0])
'mygene'
>>>print(my_list[-1])
True
```

Python: lists

Lists

The **slicing operator** is similar to the one used for strings, with the exception that for lists it can be used to modify elements.

```
>>>print(my_mylist[0:2])
['mygene', 6.16e-10]

>>>my_list[4] = False
>>>print(my_list)
['mygene', 6.16e-10, 0.0006, 3, False]
```

Every slicing operation **creates a new list, by coping** sliced values into the new one.
For example, the following slice return a copy of the list

```
>>>my_list_2 = my_list[:]
>>>print(my_list_2)
['mygene', 6.16e-10, 0.0006, 3, False]
>>>my_list[4] = True
>>>print(my_list)
['mygene', 6.16e-10, 0.0006, 3, True]
>>>print(my_list_2)
['mygene', 6.16e-10, 0.0006, 3, False]
```

Python: lists

Lists: operators and methods

`[lst] + [lst]` concatenation
`[lst]*nof_times` repetition (i.e. `[1,2]*3 → [1,2,1,2,1,2]`)
`len(lst)` length of a list
`lst.append(x)` append an item to the end of the list
`lst.extend(L)` append all the items of a list to another list
`lst.insert(position, x)` insert an item to a specific position

`lst.remove(x)` remove the first occurrence of the item x. Throw an error if the item is not in the list.
`lst.pop([position])` remove the last/i-th element of the list

```
lst.pop(3) # or #
del lst[3] The del operator can be used on a slice, too
# are equivalent to #
lst = lst[0:3] + lst[4:]
# but a memory usage difference arise #
```

`lst.clear()` clear the list
`lst.count(x)` count the number of occurrence of x within the list
`lst.reverse()` reverse the element of the list in place. It has not a return value, and the original list is modified.
`...`

Python: tuples

Tuples

Tuples are **immutable ordered** lists of elements.

```
>>>my_tuple = ('mygene', 6.16e-10, 0.0006, 3, False)
>>>print(my_tuple)
('mygene', 6.16e-10, 0.0006, 3, False)
```

A tuple has some of the methods and operators of lists, except those for modifying it since tuples are immutable.

```
>>>print(my_tuple[0])
'mygene'
>>>print(my_tuple[0:2])
('mygene', 6.16e-10)
```

Python: copy

Shallow copy, deep copy and references/pointers

We can make **lists of tuples**

```
>>>my_list = [ (1,2), (3,4), (5,6)  ]
```

And of course we can make **lists of lists**

```
>>>my_list = [ [1,2], 3, 4 ]
```

However, tuples are more similar to primitive data types and they are ever copied.
On the contrary, **lists are pointers/references**.

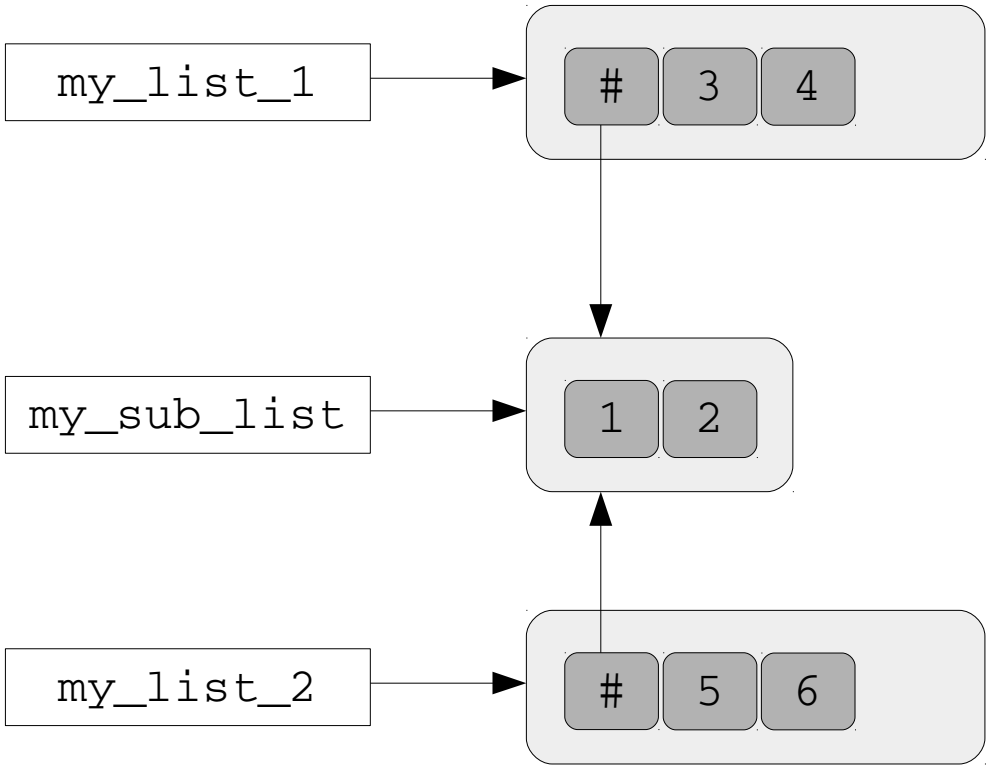
```
>>>my_sub_list = [1,2]
>>>my_list_1 = [ my_sub_list, 3, 4]
>>>my_list_2 = [ my_sub_list, 5, 6]

>>>my_sub_list[0] = 0
>>>print(my_list_1)
[[0,2],3,4]
>>>print(my_list_2)
[[0,2],5,6]
```

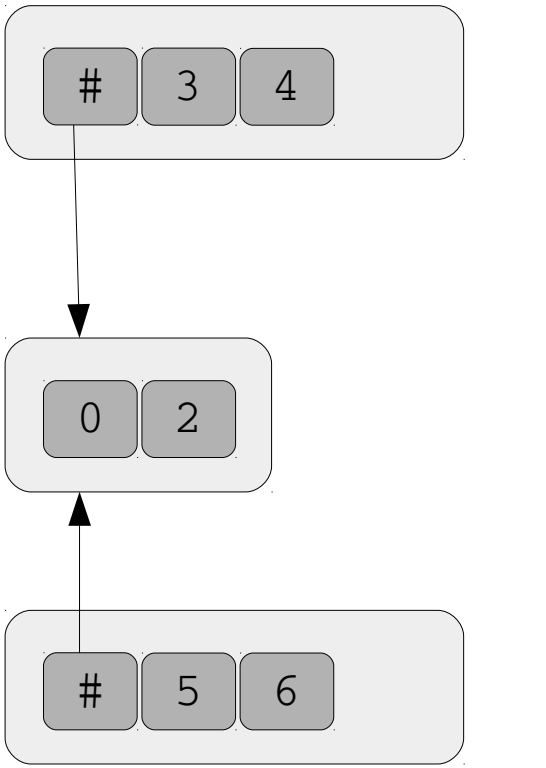
Python: copy

Shallow copy, deep copy and references/pointers

```
>>>my_sub_list[0] = 0
>>>print(my_list_1)
[[0,2],3,4]
>>>print(my_list_2)
[[0,2],5,6]
```




```
# after #
my_sub_list[0] = 0
```



Python: copy

Shallow copy, deep copy and references/pointers

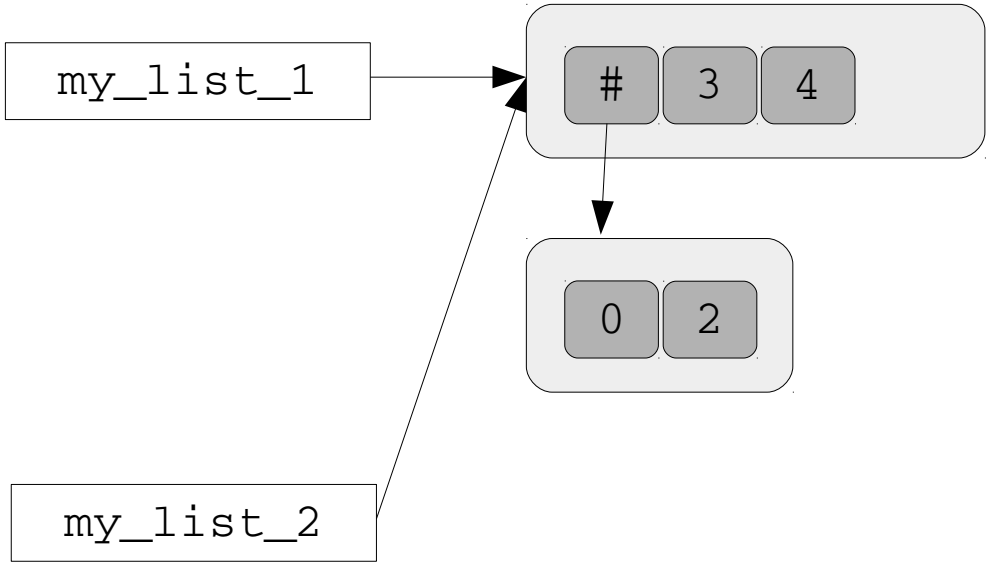
The default copy operator (i.e. `lst.copy()` or `copy(lst)`) is a shallow copy, this means that it only copies pointers but it does NOT recursively copy their pointed data.

```
>>>my_list_1 = [ [1,2], 3, 4]
>>>my_list_2 = my_list_1

>>>my_list_1[0][0] = 0

>>>print(my_list_1)
[[0,2],3,4]

>>>print(my_list_2)
[[0,2],3,4]
```



Python: copy

Shallow copy, deep copy and references/pointers

The default copy operator (i.e. `lst.copy()` or `copy(lst)`) is a **shallow copy**, this means that it only copies pointers but it does NOT recursively copy their pointed data.

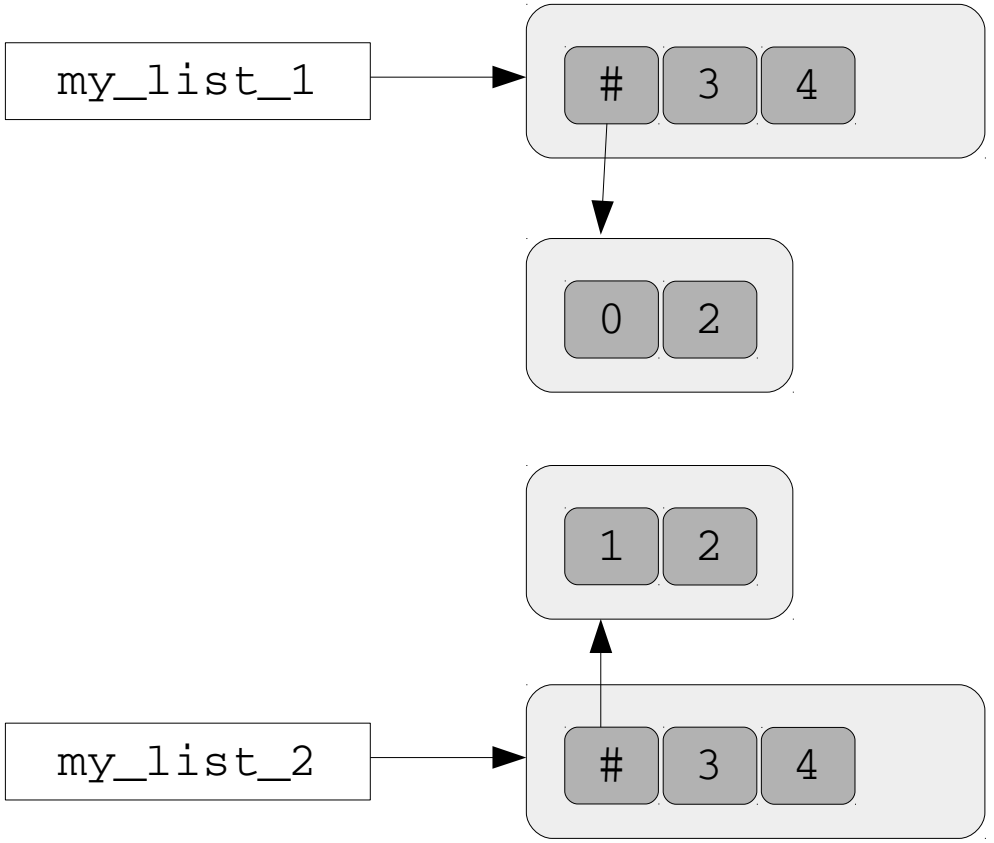
Python provides an additional copy operator for **deep coping.**

```
>>>from copy import deepcopy
>>>my_list_2 = deepcopy(my_list_1)

>>>my_list_1[0][0] = 0

>>>print(my_list_1)
[[0,2],3,4]

>>>print(my_list_2)
[[1,2],3,4]
```



Python: sets

Sets

A set is an **unordered collection** of item that does not allow for duplicates.

```
>>>my_set = {'DNA repair', 2, 0.003}
>>>print(my_set)
{'DNA repair', 2, 0.003}

>>>my_set = set()
>>>my_set.add('DNA repair')
>>>my_set.add(2)
>>>my_set.add(0.003)
>>>print(my_set)
{'DNA repair', 2, 0.003}

>>>my_set.remove(0.003)
>>>print(my_set)
{'DNA repair', 2}
```

Python: sets

Set operators

Python provides specific set theoretic operators.

```
>>>A = { 1, 2, 3}
>>>B = { 1, 4, 5}
```

```
>>>A | B
{1, 2, 3, 4, 5}
```
Union (AND)

```
>>>A & B
{1}
```
Intersection (OR)

```
>>>A - B
{2, 3}
```
Difference

```
>>>A ^ B
{2, 3, 4, 5}
```
Symmetric difference (XOR)

All of the previous operators return a new set, but `&=`, `|=`, `-=`, `^=` can be used to modify `A`.

`A <= B`	test if every element of A is in B
`A >= B`	and vice versa
`1 in A`	return `True` iff 1 is in `A`
`1 not in A`	return `True` iff 1 is not in `A`

Python: dictionaries

Dictionaries

A dictionary is an **unordered set of key and value pairs**. Namely, it provides **a mapping from keys to values**. Within a dictionary, keys are unique.

```
>>>my_dict = { 1:'one', 'two':2 }

>>>print(my_dict)
{ 1:'one', 'two':2 }
```

The **single-slice operator** can be used as for lists, namely for accessing and modifying values, but as opposite to list it can reference **non-numeric indexes** and it can be used to add a (key,value) pair.

```
>>>my_dict = dict()
>>>my_dict[1] = 'one'
>>>print(my_dict)
{1:'one'}
>>>my_dict['two'] = 2
>>>print(my_dict)
{1:'one','two':2}
```

Python: dictionaries

Dictionary methods

```
>>>my_dict = { 1:'one', 'two':2 }
```

`del my_dict['two']` remove a specific key, and its corresponding value, from a dictionary

`my_dict.get(key[, default=None])` similar to `my_dict[key]`, but if the key is not in the dictionary then return a specified default value

```
>>>my_dict.keys()
[1,'two']
```
return the list of keys of the dictionary

```
>>>my_dict.values()
[2,'one']
```
return the **list** of values in the dictionary

```
>>>my_dict.items()
[(1,'one'),('two',2)]
```
return the **list** of (key,value) pairs

Python: iterate iterables

Iteration

List, sets and dictionaries are also **iterable objects**. It means that one can iterate over they elements.

```
>>>my_list = [1,2,3]
>>>for x in my_list:
>>>    print(x, end='')
1 2 3
```

```
>>>my_set = {1,2,3}
>>>for x in my_set:
>>>    print(x, end='')
1 2 3
```

```
>>>my_dict = { 1:'one', 'two':2 }
>>>for x in my_dict.keys():
>>>    print(x, end='')
1 'two'
```

```
>>>my_dict = { 1:'one', 'two':2 }
>>>for x in my_dict.values():
>>>    print(x, end='')
'one' 2
```

```
>>>for p in my_dict.items():
>>>    print(p, end='')
(1,'one')(2,'two')
```

```
>>>for k,v in my_dict.items():
>>>    print(k,v)
1 'one'
2 'two'
```

```
# this is the equivalent code #
>>>for p in my_dict.items():
>>>    print(p[0], p[1])
1 'one'
2 'two'
```

Python: sorting

Sorting

`lst.sort(key=None, reverse=False,)` sort the list in place

`sorted(iterable[,key=None, reverse=False])` return a sorted copy of the input

Only collections (lists or sets) of **comparable items can be sorted**. If a collection contains both strings and number, a natural order among them does not exists!

```
>>>my_list = [2,4,3,1]
>>>for x in sorted(my_list):
>>>    print(x, end='')
1 2 3 4

>>>my_dict = { 1:'one', 2:'two' }
>>>For k,v in sorted(my_dict.items()):
>>>    print(k,v)
1 'one'
2 'two'
```

```
>>>my_set = {2,4,3,1}
>>>for x in sorted(my_set):
>>>    print(x, end='')
1 2 3 4
```

At the end, it is sorting the tuples `(1,'one')` and `(2,'two')`.

It can not be applied on `my_dict = { 1:'one', 'two':2 }`, because `1` and `'two'` can not be ordered.

Python: sorting

Sorting tuples

`sorted(iterable[,key=None, reverse=False])` return a sorted copy of the input

Image that you have a spreadsheet (an excel table) and you sort by columns with a specific column preference. The same can be applied to sets of tuples:

```
>>>from operator import itemgetter
>>>my_tuples={('dave', 'B', 10),('jane', 'B', 12),('john', 'A', 15)}
```

Every tuples is in the format `(name, blood, age)`

```
>>>print(sorted(my_tuples, itemgetter(2)))
[('dave', 'B', 10),('jane', 'B', 12),('john', 'A', 15)]
```

Tuples have been sorted by column 2 (column indexes start from 0), thus they are sorted by age.
The method return a list, not a set, of sorted elements.

For a comprehensive documentation on how to sort in Python3
https://docs.python.org/3/howto/sorting.html#sortinghowto

Python: list/dict comprehension

List comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
[ <generated values> <for, one or more> <filters> ]
```

Useful for **generating** list of elements

```
>>>[ i for i in range(3)]
[0,1,2]

>>>[ True for i in range(3)]
[True, True, True]
```

… having specific properties, such as “must be multiple of 2”

```
>>>[ i for i in range(10) if i % 2 == 0]
[0,2,4,6,8]
```

… or for generating combinatorial patterns

```
>>>[ (i,j) for i in range(3) for j in range(3) if i != j]
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

…

Python: list/dict comprehension

List comprehensions

... for generating nested lists

```
>>>[ [j * i for j in range(3) ]  for i in range(3)]
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

... for **extracting** elements

```
>>>my_list = [i for i in range(10)]
>>>[ i for i in my_list if i % 2 == 0]
[0,2,4,6,8]
```

Dictionary comprehensions

... for generating/extracting dictionaries

```
>>>{ i:i*2 for i in range(3)}
{0:0, 1:2, 2:4}
```

Set comprehensions

... for generating/extracting sets

```
>>>{ i for i in range(3)}
{0, 1, 2}
```

Python: functions

Functions

```
def function_name(<function parameters>):
    "string documenting the function"
    <code block>
    [return …]
```

- Passage of **parameters** to functions is almost like Java, primitive data types are passed **by copy** and other types are **passed by reference**.
- Variables declared within a function are not visible outside.
- Functions **can return more that one value**.

```
def count_AT(dna)
    """this function returns the number of occurrences of
       A and T within a DNA sequence"""
    count_a = dna.count('A') + dna.count('a')
    count_t = dna.count('T') + dna.count('t')
    return count_A, count_T

my_sequence = 'ACcccGTtGAaCGggcTTaatGAC'
count_A, count_T = count_AT(my_sequence)
help(count_AT)
```

print the documentation string of the function

Python: functions

Functions

Parameters can be declared with a **default value**.

```
def count_AT(dna, lower_case=True)
    """this function returns the number of occurrences of
       A and T within a DNA sequence"""
    count_a = dna.count('A') + dna.count('a')
    count_t = dna.count('T') + dna.count('t')
    if lower_case:
       count_a += dna.count('a')
       count_t += dna.count('t')
    return count_A, count_T

my_sequence = 'AccccGTtGAaCGggcTTaatGAC'
```

Use the **default value** (True) for the parameter `lower_case`

```
print( count_AT(my_sequence) )
```

Use a **specified value** for the parameter `lower_case`

```
print( count_AT(my_sequence, lower_case=True) )
```

If **parameter** names are used, then the **order** is not important

```
print( count_AT(lower_case=False, dna=my_sequence) )
```

Python: file I/O

Reading and writing files

The `open(filename, mode)` function is a built-in function for opening a file and writing/reading it.

For opening a file in only reading mode

```
f = open('my_file', 'r')
```

`'r'` Is the default mode, it can be omitted

```
f = open('my_file')
```

```
f = open('my_file', 'w')
```

Open the file in writing mode. If the file exists, its content is delete.

```
f = open('my_file', 'a')
```

Open the file in writing mode, but append the new content tot he end of the file

Python: file I/O

Reading and writing files

The return value of `open(filename, mode)` is an object for dealing with file operations. The most common used functions of it are:

`f.read()` Read the entire content of the file an return it as a single string.
`f.read(N)` Read N bytes form the file.
`f.readline()` Read the next line from the file.
`f.readlines()` Read the entire file and return it as a list of strings.

`f.write(string)` Write a string into the file.
`f.writelines(string_list)` Write a list of strings into a file as consecutive lines.

`f.flush()` Flushes the internal buffer.
`f.close()` Close the file streaming.

```
f_in = open('my_file_in', 'r')
f_out = open('my_file_out', 'w')
for line in f_in.readlines():
    f_out.write(line)
f_out.flush()
f_out.close()
f_in.close()
```

Python: file I/O

Reading and writing files

The object return by `open(filename, mode)` is an **iterable object.** It can be used to **read line by line a file**, without loading the entire content in memory.

```
for line in open('my_file_in', 'r'):
    print(line)
```

In the previous example the `f.close()` method is invoked automatically by python. In writing files, the `f.flush()` and the `f.close()` methods need to be called, but a special statement, the **with**-statement, facilitates the operation.

```
with open('my_file_out', 'w') as f_out:
    for line in open('my_file_in', 'r'):
        f_out.write(line)
# done #
# the with-statement manages file closing and exception handling #
```

Python: exceptions

Handling exceptions

Runtime errors are managed by **handling exception**. They can regard, for example, division by zero, opening of inexistent file of wrong file writing permission.

```
try:
    <code block that can arise an exception>
except <error type>:
    <executed if an exception is thrown>
else:
    <executed if exceptions did not arise>
finally:
    <executed anyway>
```

Try the following example with an inexistent/existent file.

```
try:
    f = open('my_file_out', 'r')
except:
    print('except')
else:
    print('else')
final:
    print('final')
```

Python: standard streams I/O

Standard streams

Python allows for reading/writing from/to standard streams, namely standard input, output and error.

```
>>>import sys
```

STDIN

```
>>>sys.stdin.read()
A line
Another line
…
# you have to press Ctrl-D to end the input #
```

Why to use it? Because it can be useful for creating
a script that can work within a pipeline.

```
my_producer_command | python3 myscript.py
```

STDOUT

```
>>>sys.stdout.write('my message to standard output')
```

STDERR

```
>>>sys.stderr.write('my message to standard error')
```

Python: modules

What are modules?
Modules are python files with the **.py extension.** They can contains definitions of **functions** oe **variables**, and are usually referred to a **specific theme**.

The `import` keyword is used to **load external modules**, namely source code written in files that are not the current script file. Firstly, import searches for files of a given name within the **current director**, then it searches within the **system path**.

Example, we can create a file named `dnautil.py` where all the functions related to handling dna sequences are located, and then we can call

```
>>>import dnautil
```

Import can be used to load all the functions within the module file, or just a selection of them, by the `from-import` statement

```
>>>from dnautil import count_AT, count_CG
```

The current system path ca be accessed by the sys module

```
>>>import sys
>>>print(sys.path)
```

And modified from a python script

```
>>>sys.path.append("a path to be appended to the system path")
```

We can also manage packages in python, that are not covered by this course
https://docs.python.org/3/tutorial/modules.html

Python: Python Standard Library

sys System-specific parameters and functions
`sys.argv` arguments passed to the current script execution
`sys.argv[0]` file name of the current executed script
`sys.argv[1]` first argument

getopt Parser for command line options
https://docs.python.org/3.5/library/getopt.html

os Miscellaneous operating system interface
Useful for dialing with files, paths, processes, etc…

math Mathematical functions
Such as `floor, ceil, factorial, log, sqrt, cos, sin, tan,` etc…

random Generate pseudo-random numbers
For generating random numbers (integers or floats) or for randomly choosing elements from lists.
`random.randint(a,b)` return a random integer between a and b (both included)
`random.choice(list)` randomly select an element from a given list
`random.shuffle(list)` randomly shuffles the given list
`random.sample(population, k)` randomly select k unique elements from a population

Python: Python Standard Library

itertools Functions creating iterators for efficient looping
https://docs.python.org/3/library/itertools.html

*The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "**iterator algebra**" making it possible to construct specialized tools succinctly and efficiently in pure Python.*

Some provided functions are the **cartesian product, permutations, combinations, combinations with replacement**, and so forth...

Python: Python Standard Library

re Regular expression operations
For searching and replacing with regular expressions, rather than find and replace methods of strings.
https://docs.python.org/3.5/library/re.html

```
import re
```
`pattern = re.compile(my_regex)` It is better to compile regex for efficiency
`pattern.search(target_string[, start, end])` search the pattern within the targeted string
`pattern.split(target_string)` split the targeted string by cutting on locations where the pattern occurs
`patter.sub(replacement, target_string)` replace the occurrences of the attern with a replacement substring

In general, regex searching methods **return a match object** if the pattern is found, else they return **None** .
The following example shows how to verify is an alias is a Ensembl ID(i.e. ENSG0001239.01).
The `re.match(regex, target_string)` can be used to compile on the fly the regex.

```
if re.match('^ENS[GTP][0-9]+(\.[0-9]+)*$', name):
    print('The name is an Ensembl ID')
else:
    print('The name is not an Ensembl ID')
```

Python: CSV files

csv CSV File reading and writing

The following code allows for reading a **TSV** file (a **CSV** file where columns are separated by tabs). For every **row** within the file, a list of values, one for each **column**, is extracted.

```
import csv
with open(myfile) as csv_file:
    reader = csv.reader(csv_file, delimiter='\t')
    for row in reader:
        print(row)
```

The following example illustrates hot to read a tab-separated file and write it into another file as a space-separated CSV.

```
import sys
ifile=sys.argv[1]
ofile=sys.argv[2]

import csv
with open(ifile) as icsv, open(ofile,'w') as ocsv:
    reader = csv.reader(icsv, delimiter='\t')
    writer = csv.writer(ocsv, delimiter=' ', quotechar='"')
    for row in reader:
        writer.writerow(row)
```

Python: CSV files

CSV file reading and writing by splitting and joining row strings

As an **alternative** to the robust CSV module, one can use split and join functions of strings.

The following example illustrates hot to read a tab-separated file and write it into another file as a space-separated CSV.

```
import sys
ifile=sys.argv[1]
ofile=sys.argv[2]
with open(ifile) as icsv, open(ofile,'w') as ocsv:
    for line in icsv:
        row = line.split('\t')
        ocsv.write( ' '.join(row)  )
```

Is it exactly the same file created by the previous example? What about escaping and quoting?

Python: named tuples and CSV

Named tuples

A named tuple is a special type of **tuple** that can be **indexed by field name**, rather that by numeric position only.

The following one is an example of how to extracted the content of a CSV file into a list of namedtuples.

```
import csv
from collections import namedtuple

with open(myfile) as csv_file:
    reader = csv.reader(csv_file, delimiter='\t')
    column_names = next(reader) # read just one row
    data_t = namedtuple('data_t', column_names)
    for row in map(data_t._make, reader):
        print(row)
        print(row.Age) # directly access fields by their name #
```

This is an example of parsed TSV file. The first line describes the column names.

```
Name   Age Sex
Ariel  21  F
Marc   22  M
Ros 18  M
Li U   23  F
```

Python: named tuples

Named tuples can be created statically, too.

```
from collections import namedtuple
```

Declaration: type name and list of fields

```
student_t = namedtuple('student_t', 'name age sex')

students = list()
students.append(student_t(name='Ariel', age=21, sex='F'))
students.append(student_t(sex='M', name='Ros', age=18))

students.append(student_t('Marc',22,'M'))
```

If the field name is not given, than the declaration order is used

```
print(students)
```

The built-in operator module provides a way to **sort** a list of namedtuple by a specific column retrieved y its name.

```
from operator import attrgetter
print(sorted(students, key=attrgetter('age')))
```

It prints a sorted copy of the original list, where students are sorted by age.

Python: OOP

Object-Oriented Programming (OOP) in Python
A **very short brief** introduction to classes made by a practical example.

The `Student.py` example file shows:

- How to declare a **class** in Python3
- The **self** keyword
- Private and public **variables**
- Static and instance **functions**
- Overriding of **built-in functions**
- **Operator overloading**
- **Sorting** custom objects

Python: external libraries

BioPython
A freely available tool collection for biological computation.
http://biopython.org/wiki/Biopython

Ploty
High-quality interactive graphing library.
https://plot.ly/python/

matplotlib
A MATLAB-like interface for 2D plotting.
http://matplotlib.org/

NumPy and SciPy
Scientific computing.
http://www.numpy.org/ http://www.scipy.org/

NetworkX
Data structures and algorithm for graphs, digraphs, and multigraphs.
https://networkx.github.io/index.html

Jupyter Notebook
An on-line interactive python enviroment, from IPython
https://jupyter.org/

Python: exercises

P.1 Write a function to check if an input string is in the genomic alphabet. It return True or False.

P.2 Write a function to check if an input string is in the amino acid alphabet. It return True in case of positive results, else it returns positions in the input string that do not respect the constraint (namely the character in that position is not a letter of the amino acid alphabet).

P.3 Write a function to calculate the CG percentage of an input genomic string. The CG percentage is give by the ration between the number of C and G and the total string length.

P.4 Write a function such that given an input genomic string it returns the reverse complement.

P.5 Write a function to verify that an input genomic string has stop codons (TGA, TAG or TAA). It must search for every possible frame-shift.

Python: exercises

P.6 Write a function for searching a string, called pattern, within a larger string, called reference. The function return every position of the reference string where the pattern occurs. DO NOT use built-in python functions such as string.find and string.count. For example, search ACCG in TTAAACCGTTATTACCGTTT and returns [4,13].

P.7 Write a function for searching subsequences by allowing gaps between pattern characters. For example, ACCG is a subsequence of ACCTTTCG with 3 gaps. For each position where the pattern sequence occurs, the function return the given position plus the number of gaps.

P.8 Write a function for reading a set of genomic sequences from a FASTA file. Remind that within a FASTA file a sequence may be split into several rows. Comment rows, starting with >, reports sequence identifiers. Use the sequences.fa file as input example.

- **P.8.1** How many sequence are in the sequence.fa file?
- **P.8.2** How long they are?
- **P.8.3** Which is the CG percentage of every string?
- **P.8.4** Which string contains the ACGT substring?
- **P.8.5** Which string contains the ACGTTGCA substring with most 3 allowed gaps?

Python: exercises

A **k-dictionary** is a set of words having the same length **k**.
A **genomic k-dctionary** is a k-dictionary of genomic words.
The D_1 dictionary identifies the **genomic alphabet** `{A,C,G,T}`.
The D_k dictionary identifies the set of all possible words of length k over the genomic alphabet, i.e. D_2=`{AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, TT}`. Words in D_k are also called **k-mers**.

P.9 Write a function for generating a D_k dictionary. The function returns the dictionary as a list of strings alphabetically sorted.

P.10 Write a function for counting the occurrences of every word of a k-dictionary. The function returns a `dict` data structure which associates to every k-mer the number of its occurrences.
P.10.1 Apply the function for k in the range [1,6] and for every sequence of the sequence.fa file.
P.10.2 Apply the function for k greater than 6. Which is the value of k such that the computational requires too much time?
P.10.3 Can you provide a different implementation for reducing the time requirement?

P.11 Write a function for generating a random genomic sequence of a given length.
P.12 Write a function for randomly shuffling a genomic sequence by translocating k-mers.