Introduction to Python

I’m good at Fortran/C, why do I need Python ?

Goal of this session:

Help you decide if you want to use python for (some of) your projects

What is Python

- Python is object-oriented (not covered today)
- Python is Interpreted (executed line by line)
    - High portability
    - Usually lower performance than compiled languages
- Python is High(er)-level (than C or Fortran)
    - Lots of high-level modules and functions
- Python is dynamically-typed and strong-typed
    - no need to explicitly define the type of a variable
    - variable types are not automatically changed (and should not)

Why Python ?

- Easy to learn
    - Python code is usually easy to read, syntax tends to be short and simple
    - The Python interpreter lets you try and play
    - Help is included in the interpreter
    - Huge community
- Straight to the point
    - Many tasks can be delegated to modules, so that you only focus on the algorithmics
- Fast
    - A lot of Python modules are written in C, so the heavy lifting is fast
    - Python itself can be made faster in many ways (there’s a session on that)
- Hugely popular

Syntax basics

Run your first program

For this tutorial you can use Jupyter (https://jupyter.org):

1. Go to https://jupyterhub.cism.ucl.ac.be (https://jupyterhub.cism.ucl.ac.be)
2. Enter your CISM credentials or ask for a temporary account in the chat
3. Click 'New' -> 'Python 3'
4. Enter `print("Hello, World !")`
5. Press `Shift + Enter`
6. Voilà !

You can also work on your laptop directly if you have Python installed

Putting it in a file

you can use your favourite text editor and enter this:

```
#!/usr/bin/env python  #tell the system which interpreter to use
print("hello world")
```

then save it as `name_i_like.py` . make it executable with:

```
chmod u+x name_i_like.py
```

and run it with:

```
./name_i_like.py
```

Python syntax 101

Assignment:

```
number = 35
floating = 1.3e2
word = 'something'
other_word = "anything"
sentence = 'sentence with " in it'
```

Note the absence of type specification (dynamic typing)

And you can do:

- `help(str)`: shows the help
- `dir(word)`: lists available methods
- `word`: displays the content of the variable

Help

Getting the help on strings:

```
In [ ]: help(str)
```

Lists

Python list : *ordered* set of *heterogeneous* objects

Assignment:

```
my_list = [1, 3, "a", [2, 3]]
```

Access:

```
element = my_list[2] (starts at 0)
last_element = my_list[-1]
```

Slicing:

```
short_list = my_list[1:3]
```

Note: slicing works like $[a, b[$: it does not include the right boundary. The example above only includes elements 1 and 2.

Dictionaries

Python dict: *ordered heterogeneous* list of (key -> value) *pairs* with very fast access

Assignment:

```
my_dict = { 1:"test", "2":4, 4:[1,2] }
```

Access:

```
my_var = my_dict["2"]
```

Missing key raises an exception:

```
In [2]: my_dict = { 1:"test", "2":4, 4:[1,2] }
        my_dict["4"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-134682133941> in <module>
      1 my_dict = { 1:"test", "2":4, 4:[1,2] }
----> 2 my_dict["4"]

KeyError: '4'
```

Flow control and blocks

An if block:

```
test = 0
if test > 0:
    print("it is bigger than zero")
else:
    print("it is zero or lower")
```

Notes:

- Control flow statements are followed by **colons**
- Block limits are defined by **indentation** (4 spaces by convention)
- Conditionals can use the `and`, `or` and `not` keywords

The for loop

The most common loop in python:

In [3]:
```
animals = ["dog", "python", "cat"]
for animal in animals:
    if len(animal) > 3:
        print (animal, ": that's a long animal !")
    else:
        print(animal)
```

```
dog
python : that's a long animal !
cat
```

Notes:

- the syntax is `for <variable> in <iterable thing>:`

For loops, continued

What if i need the index ?

```
In [4]: animals = ["dog","cat","T-rex"]
        for index, animal in enumerate(animals):
            print( "animal {} is {}".format(index,animal) )
```

```
animal 0 is dog
animal 1 is cat
animal 2 is T-rex
```

What about dictionaries ?

```
In [5]: my_dict = {"first": "Monday", "second": "Tuesday", "third": "Wednesday"}
        for key, value in my_dict.items():
            print( "the {} day is {}".format(key,value) )
```

```
the first day is Monday
the second day is Tuesday
the third day is Wednesday
```

(More on string formatting very soon)

Other flow control statements

While:

In [6]:
```python
a, b = 0, 1
while b < 100:
    print(b, end=" ")
    a, b = b, a+b # multiple assignment, more on that later
```

```
1 1 2 3 5 8 13 21 34 55 89
```

Break and continue (exactly as in C):

- `break` gets out of the closest enclosing block
- `continue` skips to the next step of the loop

Functions

In [7]:
```
def my_function(arg_1, arg_2=0, arg_3=0):
    print ("arg1:", arg_1, ", arg_2:", arg_2, ", arg_3:", arg_3)
    return str(arg_1)+"_"+str(arg_2)+"_"+str(arg_3)

my_output = my_function("a string",arg_3=7)
print("my_output:",my_output)
```

```
arg1: a string , arg_2: 0 , arg_3: 7
my_output: a string_0_7
```

Notes:

- function keyword is **def**
- arguments are passed by **reference**
- arguments can have **default values**
- when called, arguments can be given by **position** or **name**

String formatting basics

Basic concatenation:

In [8]:
```
my_string = "Hello, " + "World"
print(my_string)
```

```
Hello, World
```

Join from a list:

In [9]:
```
my_list = ["cat","dog","python"]
my_string = " + ".join(my_list)
print(my_string)
```

```
cat + dog + python
```

Stripping and Splitting:

In [10]:
```
my_sentence = " cats like mice \n    ".strip()
my_sentence = my_sentence.split() #it is now a list !
print(my_sentence)
```

```
['cats', 'like', 'mice']
```

Strings, continued

Templating:

```
In [11]: my_string = "the {} is {}"
         out = my_string.format("cat", "happy")
         print(out)
```

```
the cat is happy
```

Better templating:

```
In [12]: my_string = "the {animal} is {status}, really {status}"
         out = my_string.format(animal="cat", status="happy")
         print(out)
```

```
the cat is happy, really happy
```

The python way, with dicts:

In [13]:

```python
my_dict = {"animal":"cat", "status":"happy"}
out = my_string.format(**my_dict) #dict argument unpacking
print(out)
```

```
the cat is happy, really happy
```

f-strings

Since Python 3.6:

```
In [14]: animal = "cat"
         status = "happy"
         print(f"the {animal} is {status}, so {status}")
```

```
the cat is happy, so happy
```

You can use Python code inside the `{}`:

```
In [15]: print(f"the {animal} is {status*3}, so {status.upper()}")
```

```
the cat is happyhappyhappy, so HAPPY
```

Strings, final notes

You can specify additional options (alignment, number format)

```
In [16]: print("this is a {:^30} string in a 30 spaces block".format('centered'))
         print("this is a {:>30} string in a 30 spaces block".format('right aligned'))
         print("this is a {:<30} string in a 30 spaces block".format('left aligned'))
```

```
this is a            centered            string in a 30 spaces block
this is a                  right aligned string in a 30 spaces block
this is a left aligned                   string in a 30 spaces block
```

```
In [17]: print("this number is printed normally: {}".format(3.141592653589))
         print("this number is limited to 2 decimal places: {:.2f}".format(3.1415926535
         89))
         print("this number is forced to 6 characters: {:06.2f}".format(3.141592653589
         ))
```

```
this number is printed normally: 3.141592653589
this number is limited to 2 decimal places: 3.14
this number is forced to 6 characters: 003.14
```

The legacy syntax for string formatting is

```
"this way of formatting %s is %i years old" % ("strings", 100)
```

You'll probably see it a lot if you read older codes

# Now you know Python !

Ready for some more ?

make your life better: iPython

iPython is a shell interface to help you use python interactively instead of the Python interpreter.

It offers:

- tab completion
- history (as in bash)
- advanced help
- magic functions (for instance %timeit for benchmarking)
- calling system commands from the shell

and many other things. These are also included in Jupyter.

Unpacking

Bundle function arguments into lists or dictionaries:

```
my_list = ["dog","cat"]
my_fun(*my_list) # equivalent to 'my_fun("dog", "cat")'

my_dict = {"animal":"dog", "toy":"bone"}
my_fun(**my_dict) # equivalent to my_fun(animal="dog", toy="bone")
```

It allows to create functions with unknown number of arguments:

In [18]:
```
def my_fun(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

my_fun("pos_arg1", 34, named_arg="named")
```

```
args: ('pos_arg1', 34)
kwargs: {'named_arg': 'named'}
```

Here `args` is an unmutable list (tuple) and `kwargs` is a dictionary.

List comprehensions

Building lists:

```
In [19]: [x*x for x in range(10)]
```

```
Out[19]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Mapping and filtering:

```
In [20]: beasts = ["cat","dog","Python"]
         [beast.upper() for beast in beasts if len(beast) < 4]
```

```
Out[20]: ['CAT', 'DOG']
```

Merging with `zip`:

```
In [21]: toys = ["ball","frisbee","dead animal"]
         my_string ="the {} plays with a {}"
         [my_string.format(a,b) for a,b in zip(beasts, toys)]
```

```
Out[21]: ['the cat plays with a ball',
          'the dog plays with a frisbee',
          'the Python plays with a dead animal']
```

List comprehensions

Using an else clause:

```
In [22]: [x*x if x%3 else x for x in range(10)]
```

```
Out[22]: [0, 1, 4, 3, 16, 25, 6, 49, 64, 9]
```

Dict comprehensions work too:

```
In [23]: {x: x**2-1 for x in range(10)}
```

```
Out[23]: {0: -1, 1: 0, 2: 3, 3: 8, 4: 15, 5: 24, 6: 35, 7: 48, 8: 63, 9: 80}
```

Exercise*:

Given the following list:

```
list_of_lists = [[1,2,3,4,5], ["a","b","c","d","e"], range(5)]
```

Write a list comprehension that "reshapes" it as :

```
list_of_lists = [[1,0,"a"], [2,1,"b"],...]
```

can you find a shorter solution to get:

```
list_of_lists = [[1,"a",0], [2,"b",1],...]
```

```
In [24]: lol = [[1,2,3,4,5], ["a","b","c","d","e"], range(5)]
```

Solution

```
In [25]: list_of_lists = [[1,2,3,4,5], ["a","b","c","d","e"], range(5)]
```

```
In [26]: lol_1 = [[i[0], i[2], i[1]] for i in zip(*list_of_lists)]
         print(lol_1)
```

```
[[1, 0, 'a'], [2, 1, 'b'], [3, 2, 'c'], [4, 3, 'd'], [5, 4, 'e']]
```

```
In [27]: lol_2 = list(zip(*list_of_lists))
         print(lol_2)
```

```
[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), (4, 'd', 3), (5, 'e', 4)]
```

Reading files (basics)

open a text file for reading:

```
f = open("myfile.txt")
```

f is a **file descriptor**

Reading one line at a time:

```
line = f.readline()
```

readling the whole file to a list of lines:

```
lines = f.readlines()
```

Dealing with files : the proper way

Python offers a nicer way to read a file line by line:

In [28]:
```python
with open("houses.csv") as f:
    for line in f:
        print(line.strip())
```

```
"uid","house"
4,"kitch world"
0,"dog house"
1,"hope of getting rid of you"
5,"upside down"
2,"grass"
3,"Cretaceous"
```

Explanation:

- the **with** keyword starts a **context manager**: it deals with opening the file and executes the block only if it succeeds, then closes the file.
- file descriptors are iterable (line by line)

My favourite python tricks

You probably don't need regular expressions:

```
In [29]: my_string = "The cat plays with a ball"
         if "cat" in my_string:
             print("found")
```

```
found
```

this works on lists too:

```
In [30]: my_list  = [1,1,2,3,5,8,13,21]
         if 8 in my_list:
             print("found")
```

```
found
```

and on dictionary keys (very fast):

```
In [31]: my_dict = {"cat":"ball", "dog":"bone"}
         if "python" in my_dict:
             print("found")
```

Favourites 2

- Everything is True or False:

```
In [32]:
my_list = []
if my_list:
    print("Not empty")

my_string = ""
if my_string:
    print("Not empty")
```

In general, empty iterables are False, non-empty are True

- The useful and very readable ternary operator:

```
In [33]:
test = 10
my_var = "dog" if test > 15 else "cat"
print(my_var)
```

```
cat
```

Favourites 3

Not sure if a key exists in a dictionary ? use get

In [34]:
```
my_dict = {"cat":"ball", "dog":"bone"}
print(my_dict.get("python","default toy"))
```

```
default toy
```

Multiple assignment works as expected:

In [35]:
```
a = "python"
b = "dog"
a, b = b, "cat"
print(a, b)
```

```
dog cat
```

You can use it to make functions that return multiple values:

In [36]:

```
def my_function():
    return "cat", "dog"
var_a, var_b = my_function()
print(var_a, var_b)
```

```
cat dog
```

Favourites 4: on lists

Sort and reverse lists:

```
In [37]:
animals = ["dog","cat","python"]
for animal in reversed(animals):
    print(animal, end=" ")
print("\n---")
for animal in sorted(animals):
    print(animal, end=" ")
```

```
python cat dog
---
cat dog python
```

note: sorted takes an optional "key" argument to tell it how to sort.

quick checks on lists:

```
In [38]:
list = ["cat", "dog", 0, 6]
print(any(list)) # if at least one element is "True"
print(all(list)) # if all elements are "True"
```

```
True
False
```

Python variables explained

All Python variables are **references** a.k.a labels to objects.

When you do:

```
a = [1, 2, 3]
b = a
```

then `a` and `b` are both references for the same in-memory object (the `[1,2,3]` list). So if you do:

```
In [39]: a = [1, 2, 3]
         b = a
         a[1] = 5
         print(b)
```

```
[1, 5, 3]
```

then you have changed the object labelled by both a and b !

Python variables

Be cautious though: **assignment** (using `=`) creates a new label and **replaces** any existing label with that name:

In [40]:
```
a = [1, 2]
b = a
a = [3, 4]
print("a =", a, "and b =", b)
```

```
a = [3, 4] and b = [1, 2]
```

This does not make `b = [3, 4]`, as the `b` label is still attached to `[1, 2]`. It only creates a new label `a` attached to `[3, 4]`.

Python variables: pitfalls

The combination of this and the **local scope** of variables in functions can lead to unintuitive behaviours:

In [41]:
```
def my_func(my_list):
    my_list[0] = 3

my_list = [0, 1, 2]
my_func(my_list)
print(my_list)
```

```
[3, 1, 2]
```

modifies the input parameter as expected. However:

In [42]:
```
def my_func(my_list):
    my_list = my_list + [3]

my_func(my_list)
print(my_list)
```

```
[3, 1, 2]
```

this assignment defines a **local** `my_list` variable which **overrides the reference** in the scope of the function: it has no effect on the `my_list` argument.

# Modules and Packages

Modules

Modules allow you to use external code (think "libraries")

use a module:

```
import csv
help(csv.reader)
```

or just part of it:

```
from csv import reader
help(reader)
```

just don't import everything blindly:

```
from csv import *  # this is dangerous, can you guess why ?
```

Module example : csv

csv is a **core module**: it is distributed by default with Python

In [43]:
```
import csv
with open('my_file.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print("row:", row)
        print("the {animal} plays with a {toy}".format(**row))
```

```
row: {'animal': 'dog', 'toy': 'bone'}
the dog plays with a bone
row: {'animal': 'cat', 'toy': 'ball'}
the cat plays with a ball
```

- `DictReader` is an object from the csv package
- `reader` is an iterator built by DictReader
- `reader` gives dictionaries, for instance `{"animal":"dog", "toy":"bone"}` and affects them to the `row` reference
- keys names are taken from the first line of the csv file

writing csv files

Writing is similar:

In [44]:
```
import csv
with open('my_file_2.csv', 'w') as csvfile: # open in write mode
    writer = csv.DictWriter(csvfile, fieldnames=['animal', 'toy'])
    writer.writeheader()
    writer.writerow({'animal': 'cat', 'toy': 'laptop'})
    writer.writerow({'animal': 'dog', 'toy': 'cat'})
```

In [45]:
```
! cat  my_file_2.csv # linux command to show content of file
! rm my_file_2.csv
```

Installing modules

The standard package manager is **pip**:

- Search for a package:

```
pip search BeautifulSoup ← famous html parser
```

- Install a package:

```
pip install BeautifulSoup  # use "--user" to install in home
```

- Upgrade to latest version:

```
pip install --upgrade BeautifulSoup
```

- Remove a package:

```
pip uninstall BeautifulSoup
```

Working in a protected environment

Sometimes you need specific versions of modules, and these modules have dependencies, and these dependencies conflict with system-wide packages, etc.

In these cases you should use the `virtualenv` package:

```
pip install virtualenv # install the package, only once
virtualenv my_virtualenv
source my_virtualenv/bin/activate
```

You can then use pip to install anything you need in this virtualenv and do your work. Finally:

```
deactivate
```

closes the virtualenv session. Packages you have installed in it are not visible anymore.

Exceptions

Exceptions handling

Basics: `try` and `except`

In [46]:
```
my_var = "default animal"
my_dict = {}
try:
    my_var = my_dict["animal"]
except KeyError as err:
    print("a key error was raised for key : {}".format(err))
    print("the key 'animal' is not present")
```

```
a key error was raised for key : 'animal'
the key 'animal' is not present
```

Note: there's a far better solution for this specific problem

Ask forgiveness, not permission

Python styling recommends to avoid "if" and use exception handling instead.

Here is an (exaggerated) ugly and dangerous example:

In [47]:
```
import os
if (os.path.isfile("file_1.txt")):
    f1 = open("file_1.txt")
    if(os.path.isfile("file_2.txt")):
        f2 = open("file_2.txt")
```

(We'll discuss the "os" module later)

Ask forgiveness, not permission (II)

The Python way of dealing with this would be:

In [48]:
```
try:
    f1 = open("my_file.csv")
    f2 = open("my_file2.csv")
except IOError as io:
    print("Input file error : {}".format(io))
else:
    pass # do some stuff with f1 and f2
```

```
Input file error : [Errno 2] No such file or directory: 'my_file2.csv'
```

- The code is more flat/readable
- Errors are well-separated and handled together
- Errors are reported properly

# Coding for the future

Commenting your code

The basic comment is simply

```
# this is a comment
```

But if you think it's useful, you should make it public like this:

In [49]:
```
def my_function():
    """
    This is the help for my_function:
       it does stuff
    """
    pass
```

this way I can do:

In [50]:
```
help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
    This is the help for my_function:
       it does stuff
```

Including self-tests

the simplest way to include checks is the doctest package: let's say you have:

In [51]:
```
def plusone(x):
    """ add 1 to input parameter """
    return x+1
```

in "my_file.py". You just need to write a "my_file_test.txt" file with:

```
>>> from my_file import plusone
>>> plusone(4)
5
```

and then you can do:

```
python -m doctest test.txt # use -v for detailed output
```

It will run the lines in the `test.txt` file and check the outputs.

Proper logging

Your program will have different levels of verbosity depending if you are in test, beta or production phase. In order to avoid commenting and uncommenting "print" lines, use logging:

```
import logging
logging.basicConfig(level=logging.WARNING)
logging.warning('something unexpected happened')
logging.info('this is not shown because the level is WARNING')
```

You can also redirect the output to a file with:

```
logging.basicConfig(filename='example.log')
```

Importing scripts for debugging

You know you can import any file as a module. This allows to debug in the interpreter by using:

```
import my_file
```

to access functions and objects. But if you do this the main code itself will run !

You can avoid that by putting the code to be executed only when the script is run (not imported) inside a block like this:

```
def my_function():
    ...

if __name__ == '__main__': # that's two underscores
    print(my_function())   # put main code here
```

That way the "print" will not be called when you import my_file.

Write good code

- Have a look at PEP8 too to make your code pretty and readable: https://www.python.org/dev/peps/pep-0008 (https://www.python.org/dev/peps/pep-0008)
- Read the Zen of Python:

# Modules you need

Interacting with the OS and filesystem:

- sys:
    - provides access to arguments (argc, argv), useful sys.exit()
- os:
    - access to environment variables
    - navigate folder structure
    - create and remove folders
    - access file properties
- glob:
    - allows you to use the wildcards * and ? to get file lists
- argparse:
    - easily build command-line arguments systems
    - provide script usage and help to user

Enhanced versions of good things

- itertools: advanced iteration tools
    - cycle: repeat sequence ad nauseam
    - chain: join lists or other iterators
    - compress: select elements from one list using another as filter
    - ...
- collections: smart collections
    - defaultDict: dictionary with default value for missing keys (powerful!)
    - orderedDict: you know what it does
    - Counter: count occurrences of elements in lists
    - ...
- re: regular expressions
    - because honestly "in" is not always enough

Utilities

- copy:
    - sometimes you don't want to reference the same object with a and b
- time:
    - manage time and date objects
    - deal with timezones and date/time formats
    - includes time.sleep()
- pickle:
    - allows to save any python object as a string and import it later
- json:
    - read and write in the most standard data format on the web
- urllib:
    - access urls, retrieve files

Python 2(.7) vs python 3(.8)

Python 3+ is now recommended but many codes are based on python 2.7, so here are the main differences (2 vs 3):

- print "cat" vs print("cat")
- 1 / 2 = 0 vs 1 / 2 = 0.5
- range is a list vs range is a generator
- all strings are unicode in python 3

There's a lot more, but that's what you will need the most

Exercise

you will find 3 csv files in /home/cp3/jdf/training (/home/ucl/cp3/jdefaver/training on HMEM):

1. List files
2. read each file using the csv module
3. as you read, build a dictionary of dictionaries using the id as a key, in the form:

```
{
    0: { 'animal':'dog', 'toy':'bone', 'house':'dog house' },
    1: { 'animal':'cat', ... },
    ...
}
```

1. write one line per id with the format:

```
"the <> plays with a <> and lives in the <>"
```

```
In [ ]:
```

Exercise: going deeper

Pick any exercise below:

- write the result in a csv file
- what if one csv file was on a website ?
- write output to screen as a table with headers
- allow to switch to a html table using arguments
- allow for missing ids in one of the files
- How could you make your script shorter / faster ?

In [ ]:

```

```