## Introduction to Python

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

Jérôme de Favereau (IRMP/CP3)

jerome.defavereau@uclouvain.be

#### 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
- Python is Interpreted (executed line by line)
  - High portability
  - Usually lower performance than compiled languages
- Python is High(er)-level (than C or Fortran)
   o 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)

#### Syntax basics

### Your first python program

- 1. Connect to **hmem**
- 2. Enter the Python interpreter

# \$ module load Python (capital "P") \$ python

3. Enter the following function call:

#### print("hello world")

4. That's it, congratulations :)

### Putting it in a file

you can use your favourite text editor and enter this:

#!/usr/bin/env python  $\leftarrow$  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:



floating = 1.3e2

word = 'something'

other\_word = "anything"

sentence = 'sentence with " in it'

Note the absence of type specification !

And you can do: help(word), dir(word), and also word

#### Lists

Python list : ordered set of heterogeneous objects

Assignment:

my\_list = [1, 3, "a", [2, 3]]

Access:

Slicing:

### Dictionaries

Python dict : unordered heterogeneous list of (key  $\rightarrow$  value) pairs (with very fast access)

Assignment:

Access:

Missing key raises an exception:

#### 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**
- Blocks are defined by indentation (4 spaces by convention)
- conditionals can use the **and**, **or** and **not** keywords

### A for loop

The most common loop in python:

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

Notes:

• the syntax is **for <variable> in <iterable thing>**:

#### For loops, continued

What if i need the index ?

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

What about dictionaries ?

my\_dict = {0:"Monday", 1:"Tuesday", 2:"Wednesday"} for key, value in my\_dict.items(): print( "day {} is {}".format(key,value) )

(More on string formatting very soon)

#### Other flow control statements

While:

a, b = 0, 1 while b < 10: print(b) a, b = b, a+b

a, b = b, a+b  $\leftarrow$  multiple assignment, more on that later

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**

def my\_function(arg\_1, arg\_2=0, arg\_3=0): do\_some\_stuff return something

my\_output = my\_function("a\_string",arg\_3=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:

my\_string = "Hello, " + "World"

join from a list:

list = ["cat","dog","python"] my\_string = " : ".join(list)

Stripping and Splitting:

my\_sentence = " cats like mice \n ".strip() my\_sentence = my\_sentence.split() ← it is now a list !

### Strings, continued

templating:

my\_string = "the {} is {}" out = my\_string.format("cat", "happy")

better templating:

my\_string = "the {animal} is {status}, really {status}" out = my\_string.format(animal="cat", status="happy")

the python way, with dicts:

my\_dict = {"animal":"cat", "status":"happy"} out = my\_string.format(\*\*my\_dict) ← dict argument unpacking

#### Strings, final notes

• You can specify additional options (alignment, number format)

"this is a {:^30} string in a 30 spaces block".format('centered')

"this is a {:<30} string in a 30 spaces block".format('left aligned')

• 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. 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

you can probably ditch the Python interpreter and use ipython instead

### Unpacking

bundle function arguments into lists or dictionaries:

my\_list = ["dog","cat"] my\_fun(\*my\_list) → my\_fun("dog", "cat")

my\_dict = {"animal":"dog", "toy":"bone"} my\_fun(\*\*my\_dict) → my\_fun(animal="dog", toy="bone")

It allows to create functions with unknown number of arguments:

#### def my\_fun(\*args, \*\*kwargs):

here args is an unmutable list (tuple) and kwargs is a dictionary

my\_fun("pos\_arg1", 34, named\_arg="named")

#### List comprehensions

**Building lists:** 

my\_list =  $[x^*x \text{ for } x \text{ in range}(10)] \leftarrow \text{help}(\text{range})$ 

Mapping and filtering:

beasts = ["cat","dog","Python"] my\_list = [beast.upper() for beast in beasts if len(beast) < 4]

Merging:

toys = ["ball","frisbee","dead animal"] my\_string ="the {} plays with a {}" my\_list = [my\_string.format(a,b) for a,b in zip(beasts, toys)]

#### List comprehensions

Using an **else** clause:

my\_list = [x\*x if x%3 else x for x in range(10)]

**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],...]

#### List comprehensions (solution)

**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\_reshaped = [[1,0,"a"], [2,1,"b"],...]

list\_reshaped = [ [a[0], a[2], a[1]] for a in zip(\*list\_of\_lists) ]

If you want to keep the order, it's shorter:

#### list\_reshaped = [a for a in zip(\*list\_of\_lists)]

In python 3 you can even do: Ir = zip(\*list\_of\_lists)

### 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:

with open("my\_file.txt") as f: for line in f: do\_some\_stuff(line)

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:

my\_string = "The cat plays with a ball" if "cat" in my\_string:

this works on lists too:

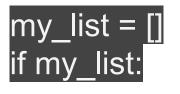
my\_list = [1,1,2,3,5,8,13,21] if 8 in my\_list:

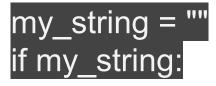
and on dictionary **keys** (very fast):

my\_dict = {"cat":"ball", "dog":"bone"} if "python" in my\_dict:

#### Favourites 2

• Everything is True or False:





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

• The useful and very readable ternary operator:

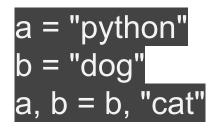
#### my\_var = "dog" if some\_condition else "cat"

#### Favourites 3

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

my\_dict = {"cat":"ball", "dog":"bone"} animal\_toy = my\_dict.get("python","default toy")

• Multiple assignment works as expected:



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

def my\_function(): return "cat", "dog" var\_a, var\_b = my\_function()

#### Favourites 4: on lists

• sort and reverse lists:

```
animals = ["dog","cat","python"]
for animal in reversed(animals):
for animal in sorted(animals):
```

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

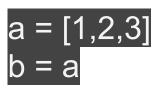
• quick checks on lists:

#### list = [i if not i%3 else 0 for i in range(10)] if any(list): ← if at least one element is "True" if all(list): ← if all elements are "True"

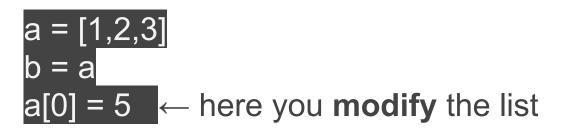
#### Python variables explained

#### All Python variables are references

if you do:



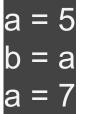
then a and b are both references (labels) for the same in-memory object (the "[1,2,3]" list). So if you do:



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

#### Python variables

Be cautious though: assignment **creates a new label** and replaces any existing label with that name:



#### ← here you **assign** a label

This **does not** make b = 7, as the "b" label is still attached to 5. It only creates a new label "a" attached to 7.

### Python variables: pitfalls

function arguments are passed by assignment...

This may look like a reference in some cases, for instance :

#### def my\_func(my\_list): my\_list[0] = 3

modifies the input parameter as expected. However:

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 ← this adds a csv **namespace** (see later) help(csv.reader)

or just part of it:

from csv import reader help(reader)

just don't import everything blindly:

from csv import  $* \leftarrow$  this is dangerous, can you guess why?

### Modules

Making modules is easy: any Python file can be a module

if I have my\_file.py with:

animals = ["cat", "dog", "python"]

I can do:

import my\_file print(my\_file.animals)

- It works the same way with **all objects**: variables, dictionaries, functions, classes, etc.
- **Packages** are bigger modules with multiple files. Making and distributing packages is very simple too.

### Module example : csv

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

import csv with open('my\_file.csv') as csvfile: reader = csv.DictReader(csvfile) for row in reader: print("the {animal} plays with a {toy}".format(\*\*row))

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

# writing csv files

writing is similar:

import csv with open('my\_file.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'})

# 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.

\$ pip install virtualenv \$ 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.

### Exceptions

# **Exceptions handling**

```
Basics:
```

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, using default")

#### do\_some\_stuff(my\_var)

Note : there's a far better solution for this specific problem, and you know it already

### Ask forgiveness, not permission

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

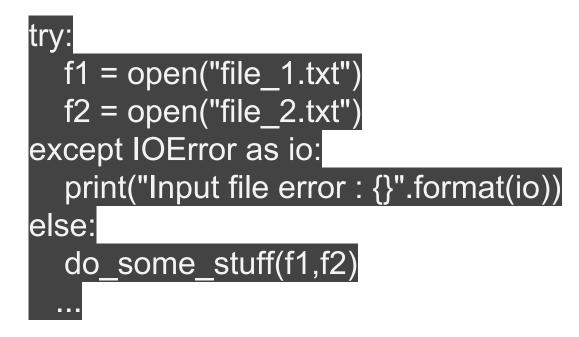
This is an (exaggerated) ugly and dangerous (why ?) example:

```
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")
    stuff = do_some_stuff(f1,f2)
```

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

# Ask forgiveness, not permission (II)

The Python way of dealing with this would be:



- 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

def my\_function(): """ describe what it does and how to use """ ← triple " do\_some\_stuff

this way if I do:

#### help(my\_function)

I'll get your nice comment directly in my interpreter

## Including self-tests

the simplest way to include checks is the **doctest** package:

def plusone(x): """ add 1 to input parameter """ return x+1

in "my\_file.py". You just need to write a "test.txt" file with:

>>> from my\_file import plusone >>> plusone(4) 5

and then:

\$ python -m doctest test.txt (use -v for detailed output)

# 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.warning('something unexpected happened') logging.info('this is not shown by default')

you can simply set the log level or target file with

logging.basicConfig(level=logging.DEBUG)

or

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 (nice !)

• but if you do this the main code itself will run !

You can avoid that by putting the "main" inside a block like this:

def my\_function(): ... ← put objects here
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

Read the Zen of Python:

>>> import this The Zen of Python, by Tim Peters

Beautiful is better than ugly. Explicit is better than implicit.

Have a look at **PEP8** too to make your code pretty and readable:

https://www.python.org/dev/peps/pep-0008/

### 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:
  - $\circ$   $\,$  allows you to use the wildcards \* and ? to get file lists
- optparse:
  - 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
  - compress: select elements from one list using another as filter

o ...

- 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

0 ...

- 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
  - o includes time.sleep()
- pickle:
  - allows to save any python object as a string and import it later
- json:
  - $\circ$   $\$  read and write in the most standard data format on the web
- urllib:
  - access urls, retrieve files

#### final comment

# Python 2(.7) vs python 3(.5)

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 an iterator
- all strings are unicode in python 3

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

#### Exercise

you will find 3 csv files in /home/ucl/cp3/jdefaver/training

- 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:...}

4. write one line per id with the format:"the <> plays with a <> and lives in the <>"

### 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
- allow for missing ids in one of the files
- How could you make your script shorter / faster ?