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

- Object-oriented (covered in another session)
- Interpreted (executed line by line)
    - High portability
    - Usually lower performance than compiled languages
- High(er)-level (than C or Fortran)
    - Lots of high-level modules and functions
- 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 things specific to your needs
- 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

Wooclap

In this sessions we will use wooclap. You can connect now at this url:

www.wooclap.com/CECIPYTHON

Or use this QR code:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Keep a tab open there, questions should automatically appear when relevant.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Basics

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Run your first program

For this tutorial you can use Jupyter:

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

You can use Python on the CECI clusters by loading the appropriate module:

```
module load Python
python
```

You can also work on your laptop 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 [1]:
```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encodin
g or
 |  errors is specified, then the object must expose a data buff
er
 |  that will be decoded using the given encoding and error hand
ler.
 |  Otherwise, returns the result of object.__str__() (if define
d)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return bool(key in self).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by
```

```
        format_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __sizeof__(self, /)
 |      Return the size of the string in memory, in bytes.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper c
ase and the rest lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless com
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |
 |  center(self, width, fillchar=' ', /)
 |      Return a centered string of length width.
 |
 |      Padding is done using the specified fill character (defa
ult is a space).
 |
 |  count(...)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of subs
tring sub in
 |      string S[start:end].  Optional arguments start and end a
re
 |      interpreted as in slice notation.
 |
 |  encode(self, /, encoding='utf-8', errors='strict')
 |      Encode the string using the codec registered for encodin
g.
 |
 |      encoding
 |        The encoding in which to encode the string.
 |      errors
 |        The error handling scheme to use for encoding errors.
 |        The default is 'strict' meaning that encoding errors r
aise a
 |        UnicodeEncodeError.  Other possible values are 'ignor
e', 'replace' and
 |        'xmlcharrefreplace' as well as any other name register
ed with
 |                          rror that can handle UnicodeEncodeErr
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
ors.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False o
therwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
 |  expandtabs(self, /, tabsize=8)
 |      Return a copy where all tab characters are expanded usin
g spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is a
ssumed.
 |
 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is foun
d,
 |      such that sub is contained within S[start:end].  Optiona
l
 |      arguments start and end are interpreted as in slice nota
tion.
 |
 |      Return -1 on failure.
 |
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions fro
m args and kwargs.
 |      The substitutions are identified by braces ('{' and
'}').
 |
 |  format_map(...)
 |      S.format_map(mapping) -> str
 |
 |      Return a formatted version of S, using substitutions fro
m mapping.
 |      The substitutions are identified by braces ('{' and
'}').
 |
 |  index(...)
 |      S.index(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is foun
d,
 |      such that sub is contained within S[start:end].  Optiona
l
 |      arguments start and end are interpreted as in slice nota
tion.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  isalnum(self, /)
 |      Return True if the string is an alpha-numeric string, Fa
lse otherwise.
 |
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |      A string is alpha-numeric if all characters in the strin
g are alpha-numeric and
 |      there is at least one character in the string.
 |
 |  isalpha(self, /)
 |      Return True if the string is an alphabetic string, False
otherwise.
 |
 |      A string is alphabetic if all characters in the string a
re alphabetic and there
 |      is at least one character in the string.
 |
 |  isascii(self, /)
 |      Return True if all characters in the string are ASCII, F
alse otherwise.
 |
 |      ASCII characters have code points in the range U+0000-U+
007F.
 |      Empty string is ASCII too.
 |
 |  isdecimal(self, /)
 |      Return True if the string is a decimal string, False oth
erwise.
 |
 |      A string is a decimal string if all characters in the st
ring are decimal and
 |      there is at least one character in the string.
 |
 |  isdigit(self, /)
 |      Return True if the string is a digit string, False other
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |
 |      A string is a digit string if all characters in the stri
ng are digits and there
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier,
False otherwise.
 |
 |      Call keyword.iskeyword(s) to test whether string s is a
reserved identifier,
 |      such as "def" or "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False o
therwise.
 |
 |      A string is lowercase if all cased characters in the str
ing are lowercase and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False oth
erwise.
 |
 |      A string is numeric if all characters in the string are
numeric and there is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
                           string is printable, False otherwise.
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |
 |      A string is printable if all of its characters are consi
dered printable in
 |      repr() or if it is empty.
 |
 |  isspace(self, /)
 |      Return True if the string is a whitespace string, False
otherwise.
 |
 |      A string is whitespace if all characters in the string a
re whitespace and there
 |      is at least one character in the string.
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False
otherwise.
 |
 |      In a title-cased string, upper- and title-case character
s may only
 |      follow uncased characters and lowercase characters only
cased ones.
 |
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False
otherwise.
 |
 |      A string is uppercase if all cased characters in the str
ing are uppercase and
 |      there is at least one cased character in the string.
 |
 |      Concatenate any number of strings.
 |
 |      The string whose method is called is inserted in between
each given string.
 |      The result is returned as a new string.
 |
 |      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
 |
 |  ljust(self, width, fillchar=' ', /)
 |      Return a left-justified string of length width.
 |
 |      Padding is done using the specified fill character (defa
ult is a space).
 |
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace remo
ved.
 |
 |      If chars is given and not None, remove characters in cha
rs instead.
 |
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given se
parator.
 |
 |      This will search for the separator in the string.  If th
e separator is found,
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js ontaining the part before the separat
```

```
or, the separator
 |      itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple contain
ing the original string
 |      and two empty strings.
 |
 |  removeprefix(self, prefix, /)
 |      Return a str with the given prefix string removed if pre
sent.
 |
 |      If the string starts with the prefix string, return stri
ng[len(prefix):].
 |      Otherwise, return a copy of the original string.
 |
 |  removesuffix(self, suffix, /)
 |      Return a str with the given suffix string removed if pre
sent.
 |
 |      If the string ends with the suffix string and that suffi
x is not empty,
 |      return string[:-len(suffix)]. Otherwise, return a copy o
f the original
 |      string.
 |
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old repl
aced by new.
 |
 |        count
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js of occurrences to replace.
```

```
 |          -1 (the default value) means replace all occurrence
s.
 |
 |      If the optional argument count is given, only the first
count occurrences are
 |      replaced.
 |
 |  rfind(...)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is fou
nd,
 |      such that sub is contained within S[start:end].  Optiona
l
 |      arguments start and end are interpreted as in slice nota
tion.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is fou
nd,
 |      such that sub is contained within S[start:end].  Optiona
l
 |      arguments start and end are interpreted as in slice nota
tion.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  rjust(self, width, fillchar=' ', /)
 |      Return a right-justified string of length width.
 |
 |      Padding is done using the specified fill character (defa
ult is a space).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given se
parator.
 |
 |      This will search for the separator in the string, starti
ng at the end. If
 |      the separator is found, returns a 3-tuple containing the
part before the
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple contain
ing two empty strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the substrings in the string, using sep
as the separator string.
 |
 |        sep
 |          The separator used to split the string.
 |
 |          When set to None (the default value), will split on
any whitespace
 |          character (including \n \r \t \f and spaces) and wil
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |          empty strings from the result.
 |        maxsplit
 |          Maximum number of splits.
 |          -1 (the default value) means no limit.
 |
 |      Splitting starts at the end of the string and works to t
he front.
 |
 |  rstrip(self, chars=None, /)
 |      Return a copy of the string with trailing whitespace rem
oved.
 |
 |      If chars is given and not None, remove characters in cha
rs instead.
 |
 |  split(self, /, sep=None, maxsplit=-1)
 |      Return a list of the substrings in the string, using sep
as the separator string.
 |
 |        sep
 |          The separator used to split the string.
 |
 |          When set to None (the default value), will split on
any whitespace
 |          character (including \n \r \t \f and spaces) and wil
l discard
 |          empty strings from the result.
 |        maxsplit
 |          Maximum number of splits.
 |          -1 (the default value) means no limit.
 |
```
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
 |      Splitting starts at the front of the string and works to
the end.
 |
 |      Note, str.split() is mainly useful for data that has bee
n intentionally
 |      delimited.  With natural text that includes punctuation,
consider using
 |      the regular expression module.
 |
 |  splitlines(self, /, keepends=False)
 |      Return a list of the lines in the string, breaking at li
ne boundaries.
 |
 |      Line breaks are not included in the resulting list unles
s keepends is given and
 |      true.
 |
 |  startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False
otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.
 |
 |  strip(self, chars=None, /)
 |      Return a copy of the string with leading and trailing wh
itespace removed.
 |
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js nd not None, remove characters in cha
```

```
rs instead.
 |
 |  swapcase(self, /)
 |      Convert uppercase characters to lowercase and lowercase
characters to uppercase.
 |
 |  title(self, /)
 |      Return a version of the string where each word is titlec
ased.
 |
 |      More specifically, words start with uppercased character
s and all remaining
 |      cased characters have lower case.
 |
 |  translate(self, table, /)
 |      Replace each character in the string using the given tra
nslation table.
 |
 |        table
 |          Translation table, which must be a mapping of Unicod
e ordinals to
 |          Unicode ordinals, strings, or None.
 |
 |      The table must implement lookup/indexing via __getitem_
_, for instance a
 |      dictionary or list.  If this operation raises LookupErro
r, the character is
 |      left untouched.  Characters mapped to None are deleted.
 |
 |  upper(self, /)
                         e string converted to uppercase.
```

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:-1]
```

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

```
 |
 |  zfill(self, width, /)
 |      Pad a numeric string with zeros on the left, to fill a f
ield of the given width.
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------------
---------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs)
 |      Create and return a new object.  See help(type) for accu
 |
 |  maketrans(...)
 |      Return a translation table usable for str.translate().
apping Unicode
 |      ordinals (integers) or characters to Unicode ordinals, s
trings or None.
 |      Character keys will be then converted to ordinals.
l length, and
 |      in the resulting dictionary, each character in x will be
 mapped to the
 |      character at the same position in y. If there is a third
 argument, it
 |      must be a string, whose characters will be mapped to Non
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
e in the result.
```

Lists

Modify:

In [2]:
```
my_list = [1, 3, "a", [2, 3]]
my_list[2] = "brol"
print(my_list)
```

```
[1, 3, 'brol', [2, 3]]
```

Multiple elements:

In [3]:
```
my_list[2:] = ["another", "list", "of", "strings"]
print(my_list)
```

```
[1, 3, 'another', 'list', 'of', 'strings']
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Dictionaries

Python dict: *ordered heterogeneous* list of (key -> value) *pairs*

Assignment:

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

Access:

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

Missing key raises an exception:

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

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[4], line 2
      1 my_dict = { 1:"test", "2":4, 4:[1,2] }
----> 2 my_dict["4"]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Dictionaries

Modify/add entry:

```
In [5]: my_dict = { 1:"test", "2":4, 4:[1,2] }
        my_dict["2"] = 5
        my_dict[2] = "five"
        print(my_dict)
```

```
{1: 'test', '2': 5, 4: [1, 2], 2: 'five'}
```

Multiple update:

```
In [6]: my_dict.update({"2": 123, 2:"six", 3:"new entry"})
        print(my_dict)
```

```
{1: 'test', '2': 123, 4: [1, 2], 2: 'six', 3: 'new entry'}
```

Wooclap

What's the result of this slicing:

```
my_var = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(my_var[-5: 8])
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Flow control and blocks

An `if` block:

```python
test = 0
if test > 0:
    print("it is bigger than zero")
elif test < 0:
    print("it is below zero")
else:
    print("it is zero")
```

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 [7]:
```
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>:`

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

For loops

A very frequent use is with the `range()` generator:

In [8]:
```
for i in range(10):
    print(i, end=" ")
print("\n---")
for i in range(10, 30, 3):
    print(i, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
---
10 13 16 19 22 25 28
```

For loops on lists

What if i need the index ?

In [9]:
```python
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
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

For loops on dicts

In [10]:
```
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)

Keys only / values only:

In [ ]:
```
for key in my_dict:
    print(key)
for value in my_dict.values():
    print(value)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Other flow control statements

While:

```
In [11]:
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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Wooclap

When will this code print "second":

```
if test > 0:
    print("first")

print("second")
```

Exercise: prime numbers

Make a list of the first natural numbers until N with `list(range(N+1))`. The goal is to replace all non-primes by `False` as in `[0, 1, 2, 3, False, 5, False, 7, False, False]`

For this:

1. start at 2
2. use a loop to mark all multiples of 2 as `False` -> `[0, 1, 2, 3, False, ...]`
3. go forward by one
    - If the number is not `False`, mark all its multiples as `False`
    - Otherwise skip it and go to the next one
4. When you reach $\sqrt{N}$ you should have "crossed" all non-primes

In [ ]:
```python
import math
N = 100
```

In [ ]:
```python
primes = list(range(N+1))
number = 2
while number < math.sqrt(N):
    if primes[number] != False:
        temp_number = number * 2
        while temp_number <= N:
            primes[temp_number] = False
            temp_number += number
    number += 1

print(*(p for p in primes if p))
```

In [ ]:

```
primes = list(range(N+1))
for number in primes[2:int(math.sqrt(N))]:
    if number != False:
        temp_number = number * 2
        while temp_number <= N:
            primes[temp_number] = False
            temp_number += number

print(*(p for p in primes if p))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [ ]:

```
primes = list(range(N+1))
for number in primes[2:int(math.sqrt(N))]:
    if number != False:
        for temp_number in range(2*number,N+1,number):
            primes[temp_number] = False

print(*(p for p in primes if p))
```

In [ ]:

```
primes = list(range(N+1))
for number in primes[2:int(math.sqrt(N))]:
    if number != False:
        primes[number*2:N+1:number] = [False]*len(primes[number*2:N+1:nu


print(*(p for p in primes if p))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [ ]:

```
import numpy as np
primes = np.arange(N+1)
for number in primes[2:int(math.sqrt(N))]:
    if number != False:
        primes[number*2:N+1:number] = False

print(*(p for p in primes if p))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Functions

```
In [ ]: 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)
```

Notes:

- function keyword is **def**
- functions can have a return value, given after the `return` keyword
- arguments can have **default values**
- arguments with default values should always come **after** the ones without
- when called, arguments can be given by **position** or **name**
- arguments called by name should always come **after** positional arguments

String formatting basics

Basic concatenation:

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

Join from a list:

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

Stripping and Splitting:

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

Strings, continued

Templating:

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

Better templating:

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

The python way, with dicts:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

f-strings

Since Python 3.6:

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

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

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

Strings, final notes

You can specify additional options (alignment, number format)

In [ ]:
```
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 align
print("this is a {:<30} string in a 30 spaces block".format('left aligne
```

In [ ]:
```
print("this number is printed normally: {}".format(3.141592653589))
print("this number is limited to 2 decimal places: {:.2f}".format(3.1415
print("this number is forced to 6 characters: {:06.2f}".format(3.141592
```

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.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Now you know Python !

Ready for some more ?

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Make your life better : use an IDE

If you plan to work on bigger projects in Python, you should consider tools to help you code faster and in a cleaner way. You should probably pick an integrated development environment (IDE). Some good (free) tools for Python are:

- Spyder
- Visual Studio Code
- PyCharm
- Sublime text

If you use VIM/Emacs, these can also be configured for most programming languages.

These tools can include:

- syntax highlight
- syntax check
- completion
- refactoring
- debugging
- versioning
- ...

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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 (like `print`):

In [ ]:

```
def my_fun(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

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

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

wooclap

What are the valid calls for the function:

```python
l = [42, "CECI"]
d = {"session": "Python"}
def my_function(arg1, arg2, session="C++", day_time="morning"):
    pass
```

- `my_function("CECI", 42, day_time="afternoon")`
- `my_function("CECI", day_time="afternoon", 42)`
- `my_function("CECI", day_time="afternoon")`
- `my_function(**d, *l)`
- `my_function(*l)`
- `my_function(*l, **d)`

In [ ]:
```python
l = [42, "CECI"]
d = {"session": "Python"}
def my_function(arg1, arg2, session="C++", day_time="morning"):
    print("OK")
```

In [ ]:
```python
my_function("CECI", 42, day_time="afternoon")
```

In [ ]:
```python
my_function("CECI", day_time="afternoon", 42)
```

In [ ]:
```python
my_function("CECI", day_time="afternoon")
```

In [ ]:
```python
my_function(**d, *l)
```

In [ ]:
```python
my_function(*l)
```

In [ ]:
```python
my_function(*l, **d)
```

List comprehensions

Building lists:

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

Mapping and filtering:

```
In [ ]: beasts = ["cat","dog","Python"]
        print([beast.upper() for beast in beasts])
        print([beast for beast in beasts if "o" in beast])
```

Merging with `zip`:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

List comprehensions

Using an else clause:

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

Double loops work too:

```
In [ ]: ["{}_{}".format(a, b) for a in ["blue", "red"] for b in ["car", "balloo
```

Dict comprehensions work too:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Wooclap

Given the following code:

```python
la = [1, 2, 3, 4, 5]
lb = ["a", "b", "c", "d", "e"]
```

which comprehension will give the dictionary:

```python
{"a": 1, "b": 2, "c": 3, "d": 4, "e": 5}
```

- `{a: b for a in la for b in lb}`
- `{b: a for a in la for b in lb}`
- `{a: b for a, b in zip(lb, la)}`
- `{a: b for b, a in zip(lb, la)}`
- `{a: b for a, b in zip(la, lb)}`

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

Simple way to search strings:

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

this works on lists too:

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

and on dictionary keys:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Favourites 2

- Everything is True or False:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Favourites 3

Not sure if a key exists in a dictionary ? use `get()`

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

Multiple assignment works as expected:

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

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

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Favourites 4: on lists

Sort and reverse lists:

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

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

quick checks on lists:

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

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

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

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 [ ]:
```
def my_func(mlist):
    mlist[0] = 3

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

modifies the input parameter as expected. However:

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

my_func(my_list)
print(my_list)
```

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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
```

Python files are modules

If you have a file called `my_module.py` with the content:

```python
my_var = "CECI"
def do_something(argument):
    pass
```

You can simply do from another file in the same folder:

```python
from my_module import my_var, do_something
new_var = my_var + " Python"
do_something(new_var)
```

The alternative syntax works too:

```python
import my_module
my_module.do_something("test_variable")
```

Making packages

Python packages are just groups of modules. To make them, you need to:

- create a folder with the name of your package
- add an empty file there called `__init__.py`
- add your module files there

For instance if I create a folder called `my_package` and add three files `__init__.py`, `first_module.py`, `second_module.py`, I can then do:

```
from my_package import first_module, second_module
print(first_module.my_first_var)
print(second_module.my_function)
```

providing that you have objects `my_first_var` and `my_function` in the respective modules.

Module example : csv

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

In [ ]:
```
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))
```

- `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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

writing csv files

Writing is similar:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Installing modules

The standard package manager is **pip**:

- 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 [ ]:
```
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")
```

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 [ ]:
```
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 [ ]:
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
```

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

Coding for the future

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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

this way I can do:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Including self-tests

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

In [ ]:
```
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.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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:

```python
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:

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

Importing scripts

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 doing this runs the whole content of `my_file.py` which is not what you want.

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, only when you run `python my_file.py`

Write good code

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

```
In [ ]: import this
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Modules you need

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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!)
  - 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
- requests:
    - access urls, retrieve remote files

Basics for science

- numpy:
  - linear algebra
  - fast treatement of large sets of numbers
- matplotlib:
  - standard library for plotting
- scipy:
  - optimization
  - integration
  - differential equations
  - statistics
  - ...
- pandas:
  - data analysis

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Python 2(.7) vs python 3(.10)

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 (Jupyterhub users) or /CECI/home/ucl/cp3/jdefaver/training (CECI users):

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', ... },
    ...
}
```

4. write one line per id with the format:

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

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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
- How could you make your script shorter / faster ?

```
In [ ]: # 1: list csv files

        import glob
        print(glob.glob('*.csv'))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [ ]:

```
# 2 put file content in a dictionary of dictionaries

my_dict = {}
with open('animals.csv') as afile:
    reader = csv.DictReader(afile)
    for row in reader:
        uid = row['uid'].strip()
        my_dict[uid] = {'animal': row['animal'].strip()}

print(my_dict)
```

In [ ]:

```python
# 3 join a second file by adding to each dict with the same uid
my_dict = {}
with open('animals.csv') as afile:
    reader = csv.DictReader(afile)
    for row in reader:
        uid = row['uid'].strip()
        my_dict[uid] = {'animal': row['animal'].strip()}

with open('toys.csv') as afile:
    reader = csv.DictReader(afile)
    for row in reader:
        uid = row['uid'].strip()
        my_dict[uid]['toy'] = row['toy'].strip()

print(my_dict)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [ ]:

```
# 4 DRY
my_dict = {}
csv_files = ['toys.csv', 'houses.csv', 'animals.csv']
for csv_file in csv_files:
    with open(csv_file) as cfile:
        reader = csv.DictReader(cfile)
        for row in reader:
            uid = row['uid'].strip()
            key = csv_file[:-5]
            if uid not in my_dict:
                my_dict[uid] = {}
            my_dict[uid][key] = row[key].strip()

print(my_dict)
```

In [ ]:
```
# 6 avoid additional checks
from collections import defaultdict

my_dict = defaultdict(dict)
csv_files = ['toys.csv', 'houses.csv', 'animals.csv']
for csv_file in csv_files:
    with open(csv_file) as cfile:
        reader = csv.DictReader(cfile)
        for row in reader:
            uid = row['uid'].strip()
            key = csv_file[:-5]
            my_dict[uid][key] = row[key].strip()

template = "the {animal} plays with a {toy} and lives in the {house}"
for _, value in my_dict.items():
    print(template.format(**value))
```

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js