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 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:

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

Basics

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 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 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 encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  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 key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
```

```
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_sp
ec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __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.
 |
 |  __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 case and th
e rest lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless comparisons.
 |
 |  center(self, width, fillchar=' ', /)
```

```
 |      Return a centered string of length width.
 |
 |      Padding is done using the specified fill character (default is a s
pace).
 |
 |  count(...)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of substring sub
in
 |      string S[start:end].  Optional arguments start and end are
 |      interpreted as in slice notation.
 |
 |  encode(self, /, encoding='utf-8', errors='strict')
 |      Encode the string using the codec registered for encoding.
 |
 |      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 raise a
 |        UnicodeEncodeError.  Other possible values are 'ignore', 'replac
e' and
 |        'xmlcharrefreplace' as well as any other name registered with
 |        codecs.register_error that can handle UnicodeEncodeErrors.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False otherwise.
 |      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 using spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is assumed.
 |
 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  format(...)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from 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 from 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 found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
```

```
 |      Raises ValueError when the substring is not found.
 |
 |  isalnum(self, /)
 |      Return True if the string is an alpha-numeric string, False otherw
ise.
 |
 |      A string is alpha-numeric if all characters in the string are alph
a-numeric and
 |      there is at least one character in the string.
 |
 |  isalpha(self, /)
 |      Return True if the string is an alphabetic string, False otherwis
e.
 |
 |      A string is alphabetic if all characters in the string are alphabe
tic and there
 |      is at least one character in the string.
 |
 |  isascii(self, /)
 |      Return True if all characters in the string are ASCII, False other
wise.
 |
 |      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 otherwise.
 |
 |      A string is a decimal string if all characters in the string are d
ecimal and
 |      there is at least one character in the string.
 |
 |  isdigit(self, /)
 |      Return True if the string is a digit string, False otherwise.
```

```
 |
 |      A string is a digit string if all characters in the string are dig
its and there
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier, False othe
rwise.
 |
 |      Call keyword.iskeyword(s) to test whether string s is a reserved i
dentifier,
 |      such as "def" or "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False otherwise.
 |
 |      A string is lowercase if all cased characters in the string are lo
wercase and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False otherwise.
 |
 |      A string is numeric if all characters in the string are numeric an
d there is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
 |      Return True if the string is printable, False otherwise.
 |
 |      A string is printable if all of its characters are considered prin
table 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 are whitesp
ace and there
 |      is at least one character in the string.
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False otherwis
e.
 |
 |      In a title-cased string, upper- and title-case characters may only
 |      follow uncased characters and lowercase characters only cased one
s.
 |
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False otherwise.
 |
 |      A string is uppercase if all cased characters in the string are up
percase and
 |      there is at least one cased character in the string.
 |
 |  join(self, iterable, /)
 |      Concatenate any number of strings.
 |
 |      The string whose method is called is inserted in between each give
n 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 (default is a s
pace).
 |
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace removed.
 |
 |      If chars is given and not None, remove characters in chars instea
d.
 |
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |
 |      This will search for the separator in the string.  If the separato
r is found,
 |      returns a 3-tuple containing the part before the separator, the se
parator
 |      itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containing the or
iginal string
 |      and two empty strings.
 |
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old replaced by ne
w.
 |
 |        count
 |          Maximum number of occurrences to replace.
 |          -1 (the default value) means replace all occurrences.
 |
 |      If the optional argument count is given, only the first count occu
```

```
rrences are
 |      replaced.
 |
 |  rfind(...)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      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 (default is a s
pace).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |
 |      This will search for the separator in the string, starting at the
end. If
 |      the separator is found, returns a 3-tuple containing the part befo
re the
```

```
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containing two em
pty strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimit
er string.
 |
 |        sep
 |          The delimiter according which to split the string.
 |          None (the default value) means split according to any whitespa
ce,
 |          and discard empty strings from the result.
 |        maxsplit
 |          Maximum number of splits to do.
 |          -1 (the default value) means no limit.
 |
 |      Splits are done starting at the end of the string and working to t
he front.
 |
 |  rstrip(self, chars=None, /)
 |      Return a copy of the string with trailing whitespace removed.
 |
 |      If chars is given and not None, remove characters in chars instea
d.
 |
 |  split(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimit
er string.
 |
 |      sep
 |        The delimiter according which to split the string.
```

```
 |        None (the default value) means split according to any whitespac
e,
 |        and discard empty strings from the result.
 |      maxsplit
 |        Maximum number of splits to do.
 |        -1 (the default value) means no limit.
 |
 |  splitlines(self, /, keepends=False)
 |      Return a list of the lines in the string, breaking at line boundar
ies.
 |
 |      Line breaks are not included in the resulting list unless keepends
is given and
 |      true.
 |
 |  startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False otherwis
e.
 |      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 whitespace r
emoved.
 |
 |      If chars is given and not None, remove characters in chars instea
d.
 |
 |  swapcase(self, /)
 |      Convert uppercase characters to lowercase and lowercase characters
to uppercase.
```

```
 |
 |  title(self, /)
 |      Return a version of the string where each word is titlecased.
 |
 |      More specifically, words start with uppercased characters and all
remaining
 |      cased characters have lower case.
 |
 |  translate(self, table, /)
 |      Replace each character in the string using the given translation t
able.
 |
 |        table
 |          Translation table, which must be a mapping of Unicode ordinals
to
 |          Unicode ordinals, strings, or None.
 |
 |      The table must implement lookup/indexing via __getitem__, for inst
ance a
 |      dictionary or list.  If this operation raises LookupError, the cha
racter is
 |      left untouched.  Characters mapped to None are deleted.
 |
 |  upper(self, /)
 |      Return a copy of the string converted to uppercase.
 |
 |  zfill(self, width, /)
 |      Pad a numeric string with zeros on the left, to fill a field of th
e given width.
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
```

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*

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'
```

Exercise

What's the result of this slicing:

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

Flow control and blocks

An if block:

```
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 [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]:
```
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

Exercise

When will this code print "second":

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

print("second")
```

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**
- named arguments should always come **after** positional arguments

String formatting basics

Basic concatenation:

In [7]:
```
my_string = "Hello, " + "World"
print(my_string)
```

```
Hello, World
```

Join from a list:

In [8]:
```
my_list = ["cat","dog","python"]
my_string = " + ".join(my_list)
print(my_string)
```

```
cat + dog + python
```

Stripping and Splitting:

In [9]:
```
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 [10]:
```
my_string = "the {} is {}"
out = my_string.format("cat", "happy")
print(out)
```

```
the cat is happy
```

Better templating:

In [11]:
```
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 [12]:
```
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 [13]:
```
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 [14]:
```
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 [15]:
```
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 [16]:
```
print("this number is printed normally: {}".format(3.141592653589))
print("this number is limited to 2 decimal places: {:.2f}".format(3.141592653589))
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. You ca use it instead of the Python interpreter in your terminal.

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.

make your life even 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
- ...

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 [17]:
```
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.

exercise

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 [18]:
```
[x*x for x in range(10)]
```

Out[18]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Mapping and filtering:

In [19]:
```
beasts = ["cat","dog","Python"]
print([beast.upper() for beast in beasts])
print([beast for beast in beasts if "o" in beast])
```

```
['CAT', 'DOG', 'PYTHON']
['dog', 'Python']
```

Merging with `zip`:

In [20]:
```
toys = ["ball","frisbee","dead animal"]
my_string ="the {} plays with a {}"
[my_string.format(a, b) for a, b in zip(beasts, toys)]
```

Out[20]:
```
['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 [21]:
```
[x*x if x%3 else x for x in range(10)]
```

Out[21]:
```
[0, 1, 4, 3, 16, 25, 6, 49, 64, 9]
```

Double loops work too:

In [22]:
```
["{}_{}".format(a, b) for a in ["blue", "red"] for b in ["car", "balloon"]]
```

Out[22]:
```
['blue_car', 'blue_balloon', 'red_car', 'red_balloon']
```

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 code:

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

which comprehension will give the dictionary:

```
{"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 [24]:
```
with open("houses.csv") as f:
    for line in f:
        print(line)
```

```
"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

Simple way to search strings:

In [25]:
```
my_string = "The cat plays with a ball"
if "cat" in my_string:
    print("found")
```

```
found
```

this works on lists too:

In [26]:
```
my_list  = [1,1,2,3,5,8,13,21]
if 8 in my_list:
    print("found")
```

```
found
```

and on dictionary keys:

In [27]:
```
my_dict = {"cat":"ball", "dog":"bone"}
if "python" in my_dict:
    print("found")
```

Favourites 2

- Everything is True or False:

In [28]:
```
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 [29]:
```
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 [30]:
```
my_dict = {"cat":"ball", "dog":"bone"}
print(my_dict.get("python","default toy"))
```

```
default toy
```

Multiple assignment works as expected:

In [31]:
```
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 [32]:
```
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 [33]:
```
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 [34]:
```
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 [35]:
```
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 [36]:
```
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 [37]:
```
def my_func(mlist):
    mlist[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 [38]:
```
def my_func(mlist):
    mlist = mlist + [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
```

Python files are modules

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

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

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

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

The alternative syntax works too:

```
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 [39]:
```
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 [40]:
```
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 [41]:
```
! 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 [42]:
```
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 [ ]:
```
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

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 [43]:
```
def my_function():
    """
    This is the help for my_function:
       it does stuff
    """
    pass
```

this way I can do:

In [44]:
```
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 [ ]:
```
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

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 [45]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
```

If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

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!)
    - 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

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

1. 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 using arguments
- How could you make your script shorter / faster ?

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

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

In [ ]:
```
# 2 read a file with csv, see that there are uids to cleanup

import csv
with open('animals.csv') as afile:
    reader = csv.DictReader(afile)
    for row in reader:
        print(row)
```

In [ ]:
```
# 3 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 [ ]:

```
# 4 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)
```

In [ ]:

```
# 5 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 [ ]:
```

```