Bases de Python

Bernard CHAMBON

CC-IN2P3, Lyon - France

12, 13, 14 novembre 2018

Rappels

- Introduction
- "Hello World" !
- Opérateurs, iteration, aiguillage, fonction,
- Chaine de caractères : concaténation, découpage, formatage
- E/S avec `stdin`/`stdout` , puis avec fichiers (avec cas du format CSV)
- Journalisation de l'exécution du code (logging)
- *Compléments (chaines de caractères avec Python 2 vs 3, module et package)*

- Caractéristiques
  langage interprété, mais avec génération de bytes-code (=.pyc)
  typage dynamique (`x = 123` , puis `x = "Hi !"`)
  gestion automatique de la mémoire
  orienté objet, héritage
  gestion des exceptions
  disponible sur toutes les plateformes (souvent python 2.7 par défaut, on est à python 3.7)
  développement rapide
  on peut l'utiliser en interactif : en ligne de commande ou dans un navigateur (via notebook)
- Inconvénient
  pas de compilateur
  nécessité d'avoir un bon éditeur (coloration syntaxique, complétion de code, liste des méthodes d'une classe, erreur de syntaxe, etc.)
  la délimitation des blocs de code se fait par indentation (à une indentation près on peut être hors d'une boucle)
  Differences Python 2 vs Python 3

On va travailler avec Python 3

- Convention de nommage - codage

```
varName = 123 ou var_name = 123

MY_CONSTANT = 123

my_great_function()

class MyAmazingClass

my_package

my_module
```

  Une ligne par import
  Fin de ligne comme marqueur de fin d'instruction (pas de ' ;' même s'il est accepté)

  voir Style Guide for Python Code

- Un lien pour de la doc svp : https://www.python.org/doc/

Rappels > Hello World !

- Premier code
  - hello_world.py

```
#!/usr/bin/env python

message =  "Hello World"
print("message" , message)

# name of the current file
print("__file__ ",__file__)

# Will be 'main' case file is executed
# Will be module name case module is imported
print('__name__ ',__name__)
```

  - Exécution ou import

Exécution du module

```
>./hello_world.py
message Hello World
__file__  /Users/bchambon/.../hello_world.py
__name__  __main__
```

Import du module

```
> python
>>> from hello_world import *

message Hello World
__file__  /Users/bchambon/.../hello_world.py
__name__  hello_world
```

- A retenir
  - shebang : pour sélection de l'interpréteur depuis $PATH
    `#!/usr/bin/env python` , PAS de chemin en dur tel ~~`#!/usr/local/bin/python`~~
  - module : un fichier de suffixe '.py', fichier contenant des instructions Python
    Peut être exécuté, ou importé dans un autre module (directive `import`), instructions aussi exécutées dans ce cas (= initialisation du module)
  - Initialisation et accessibilité de la variable `message`

- "Hello world " restructuré

Code

```
#!/usr/bin/env python

import sys

##
def hello_world():
  message = "Hello You Again !"
  print(message)

##
def main():
  hello_world()
  return(0)

##
if __name__ == '__main__':
  # return code (rc) used as exit code for shell
  rc = main()
  sys.exit(rc)
```

Exécution ou import du module

Exécution

```
>./hello_world.py
Hello You Again !
```

Import du module

```
>python
Python 3.6.3 |Anaconda, Inc.| ...
>>> from hello_world import *
>>>
>>> hello_world()
Hello You Again !
>>>
```

- A retenir
  - Indentation pour marquer un bloc, programmation avec des fonctions, return code
  - `print(value)` et non pas ~~`print value`~~ , pour avoir du code compatible python 3
  - `__name__` utilisé pour définir le point d'entrée de l'exécution d'un module (cf `main` de java)
  - les codes qui suivent adopteront ce style

- Opérateurs
  - Numériques
    `+`, `-`, `*`, `/`, `//` (quotient dans division entière), `%` (reste), `+=`, `-=`, `*=`, `/=`, `**` (puissance)
    `/` donne un entier en Python 2 (`3/2` donne `1`), un réel en Python 3 (`3/2` donne `1.5`)
    Pas de `x++` ni de `x-`
  - Logiques
    `and`, `or`, `not`
  - Comparaisons
    `<`, `<=`, `>`, `>=`
    `==` même contenu, `!=` , `is` même object, `is not`
    `in` existance dans une collection
  - Quelques valeurs prédéfinies
    `True`, `False`
    `None` (ex `value=None` , `if value is None:`). Idem `NULL` en C++, `nul` en Java
    `None` peut aussi être une instruction(⇔ No operation)
    `pass` : instruction qui ne fait rien (mais peut permettre de créer une fonction ou une classe vide)
  - Chaines de caractères
    `+` pour concaténation(ex : `"hello " + "world"`)
    conversion chaine ↔ numérique : `str(123)` , `int("123")` , `float("123.45")`
  - `==` vs `is`

```
>>> a=[1, 2, 3]
>>> b=[1, 2, 3]
>>> print(a == b, a is b)
(True, False)
```

- Itération
  - `for`

```
# range in python 2 or 3 (xrange in python 2)
for i in range(0, 5+1):  # for value between [0 and 5]
  print(i)
```

  - `while`

```
i=0
while (i<=5):
  print(i)
  i+=1
```

  - Pas `do - while`
  - `break` et `continue`
    `break` pour arrêter l'itération, `continue` pour continuer l'exécution à l'itération suivante
- Aiguillage
  - `if else , if elif elif ... else`

```
if (x < 100):  # note the colon ':' mark
  print("tens" )
elif (x < 1000):
  print("hundreds")
else :
 print("thousands")

# if/else also available in one line
msg = "even" if (x%2 == 0) else "odd"
```

  - Pas de `switch - case`

- Syntaxe, variable locale | globale, valeur de retour, argument par défaut
  - `def`
    `def` débute une fonction (nom en snake_case par convention), suivi de `:`
    Fin de fonction selon l'indentation
  - `return`
    Pour spécifier ce qui sera retourné par la fonction, il est même possible de renvoyer plusieurs valeurs

```
def do_compute():
  # ...

  # case error, returning error code and error message
  error_code=123;
  error_msg="Failure while processing request ..."
  return (error_code, error_msg)

# using do_compute
rc, msg = do_compute()
```

    Pour une fonction qui ne renvoi rien, `return` n'est pas nécessaire. En général il n'est pas mis
  - Variable
    > variable déclarée dans la fonction a une portée locale. Si déclarée dans un module, portée globale (= visible dans toutes les fonctions du module)
    Pour affecter une valeur à une variable de portée globale, il faut utiliser le mot clé `global`
    > possibilité d'arguments par défaut, ils doivent être en fin de liste des arguments
  - Une fonction très utilisée : `print`
    `print` peut être vu comme une instruction avec Python2, mais obligatoirement comme une fonction avec Python 3
    `print("Hello")` et non pas ~~`print "Hello"`~~

- Syntaxe par l'exemple

Code

```
#!/usr/bin/env python

import sys

### definition with default parameters
def hello(message="Hello", to="You"):
  print(message, to)
  return

## usage of function
def main():
  hello()
  hello(to="World")
  hello("Nice to meet")  # idem hello(message = "Nice to meet")
  return(0)

##
if __name__ == '__main__':
  rc = main()
  sys.exit(rc)
```

Exécution

```
Hello You
Hello World
Nice to meet You
```

- Notation, accès
  - Notation
    Délimitée par soit ' ' ou " " soit """ """ ou ''' ''' si sur plusieurs lignes
    Une chaine est de type `str` , voir `print(type("hello"))`
    Pas de méthode `.len()`, mais la fonction `len(...)`

    Pour plus de précision sur les chaines (str, unicode, bytes, encodage ascii | utf-8, Python 2 vs 3)
    Voir le complément en fin de présentation

  - Accès aux caractères ou à une sous-chaine
    > en considérant comme un tableau de caractères, donc via `[i:j:step]`

```
message = "Python is your friend"
print(message[0])                # will be P
print(message[0:6])              # will be Python
print(message[0:len(message)])   # will be Python is your friend
print(message[-1])               # will be last letter 'd'
```

    > en sachant que la chaine est itérable caractère / caractère tel que `for c in message :`

```
message = "Python is your friend"
for c in message:
  sys.stdout.write(c)  # will give  'Python is your friend'
```

    > en utilisant les méthodes disponibles de la classe str

  - Test d'un chaine vide par l'opérateur `not`

```
message = ""
if not message:
  print("message is None or empty")
```

- Concaténation, découpage, formatage
  - Concaténation
    > via l'opérateur `+`

    ```
    message="Hello" + " World"
    ```

    > via le symbole `% tel que %s, %d, %f`

    ```
    message=("%s %s")%("Hello", "World")
    ```

    > via la méthode `format()` et avec le symbole `{}`

    ```
    message = "{} {}".format("Hello", "World")
    ```

  - Découpage
    via la méthode `split([sep[, maxsplit]])`, qui renvoi une liste

    ```
    # will give ['Python', 'is', 'your', 'friend']   # Notice brackets indicating a list
    alist = "Python is your friend".split()

    # will give ['Python', 'is', 'your friend']
    alist = "Python is your friend".split(" ", 2)
    ```

- Formatage
  - Syntaxe par l'exemple
    `%` en style classique ou méthode `format()` + `{}` en style moderne
    > style classique : `%`

```
"%s" (chaine) "%d" (entier)  "%8d" (entier sur 8c)  "%-8.2f" (aligné à gauche, 8 c dont 2 après la virgule)
PI=3.141592653
print("%-+10.5f"%(PI))  # first '-' for align, '+' next to number for sign
+3.14159
print("%+10.5f"%(PI)) # default is right align, '+' next to number for sign
  +3.14159
```

    > style moderne (et plus riche) : `format()` + `{}`

```
print("{:>+10.5f}".format(PI))
  +3.14159

print("{:_<10} {:_^10} {:_>10}".format("Welcome", "to", "Python"))
Welcome___ ____to____ ____Python
```

  - Plus de détails
    https://pyformat.info/

- # ou """ """
  - #
    Pour un commentaire associé à une ou quelques instructions
    fin de commentaire à la fin de la ligne

```
# now let's computed minimal value
value = ...
```

  - Les docstrings via """ ... """
    Pour commenter une fonction, une classe, un module

```
def do_compute() :
   """ This function compute value according to the
       new algorithm based on great work from ...
   """
   ...
   return(value)
```

    Noter que le commentaire est mis au début de la fonction (ou dans la classe)
    Il est accessible via la l'attribut magique `__doc__`, on peut ainsi écrire `print(do_compute.__doc__)`
    En interactif, il est accessible via `?` ou `help(...)`
    Dans notre cas via `?do_compute` ou `do_compute?` ou `help(do_compute)`
    Il existe des outils d'extraction de commentaires docstrings (cf pydoc)
  - Et pour info ...
    Les docstrings peuvent permettre de spécifier des tests, qui seront exécutés via `doctest` (`python -m doctest ...`) - Attention à la lourdeur et la lisibilité

Rappels > E/S sur stdin/stdout|err

- Lecture depuis stdin
  - `stdin.readline()` (Python 2 | 3)
    pour lecture jusqu'au CRLF inclus, utiliser `.rstrip()`
    renvoi une chaine vide sur fin de fichier (ctrlD si saisie clavier)
  - `raw_input([prompt])` (Python 2.x)
    pour lecture jusqu'au CRLF non inclus
  - `input([prompt])` (Python 3.x)
    pour lecture jusqu'au CRLF non inclus
  - Cas de lecture de plusieurs lignes
    `sys.stdin` renvoi un iterable
    lecture ligne par ligne jusqu'à fin de fichier (ctrlD si saisie clavier), ou lecture jusqu'à un mot clé
- Ecriture sur stdout | stderr
  - `print()`, qui est une fonction !
    par défaut vers stdout et avec un saut de ligne, mais avec python 3 (ou python 2.7 + import ad-hoc) on peut faire plus
    exemple : `print(message, end='', file=sys.stderr, flush=True)`
  - `write()`, méthode de `stdout | stderr`
    `stdout | stderr.write(astring)`
  - `flush()`, méthode de `stdout | stderr`
    `stdout | stderr.flush()`

- Syntaxe par l'exemple

Code utilisant `readline()` ou `input()`

```
def read_oneline():
  msg1 = sys.stdin.readline()
  print("With readline, you get '%s'" %(msg1))
  print("but after rstrip(), you get '%s'" %(msg1.rstrip()))

  # specific to Python 2.x
  # msg2 = raw_input("\nEnter message > ")
  # print("you enter message '%s'" %(msg2))
  #

  # specific to Python 3.x ,
  msg3 = input("\nEnter another message > ")
  print("you enter another message '%s'"%(msg3))
```

Exécution

```
azerty
With readline, you get 'azerty
'
but after rstrip(), you get 'azerty'

Enter another message > Welcome again !
you enter another message 'Welcome again !'
```

- Syntaxe par l'exemple

Code utilisant un iterateur sur `sys.stdin`

```
# Version using sys.stdin as iterable
def read_lines_until_eof():
  print("begin of input" )
  for line in sys.stdin:
    print("you enter the line '%s'" %(line.rstrip()))
  print("end of input" )


# Other version using sys.stdin.readline()
def read_lines_until_eof_v2():
  print("begin of input")
  line = sys.stdin.readline()
  while (line):
    print("you enter the line '%s'" % (line.rstrip()))
    line = sys.stdin.readline()
  print("end of input")
```

Exécution

```
begin of input
message
you enter the line 'message'
in a bottle
you enter the line 'in a bottle'
^D
end of input
```

- Redirections des `stdout` | `err`
  - Il est possible de re-diriger, de manière temporaire, les `stdout` | `err` vers un fichier ou un object
    Utile si on ne maitrise pas un code verbeux !
  - Syntaxe par des exemples

- Syntaxe par l'exemple : redirection de `stdout` vers un fichier

Code

```
def too_verbose():
  for i in range(0, 10):
    print(i)

def main():
  print("Start running task ...")
  sys.stdout.flush()
  save_stdout = sys.stdout
  sys.stdout = f = open('/tmp/too_verbose.output', 'w')

  # calling the too verbose method
  too_verbose()

  f.close()
  sys.stdout = save_stdout
  print("End running task")
```

Exécution

```
Start running task ...
End running task


> more /tmp/too_verbose.output
0
1
2
...
```

- Syntaxe par l'exemple : redirection de `stdout` vers un object `StringIO`

Code

```
# Caution StringIO class from
#  package 'cStringIO' (Python 2) or 'io' (Python 3)

# from cStringIO import StringIO # Python 2
from io import StringIO  # Python 3
...

def main():
  print("Start running task ...")
  sys.stdout.flush()
  save_stdout = sys.stdout
  sys.stdout = mystdout = StringIO()

  too_verbose()

  sys.stdout = save_stdout


  print("End running task")

  print("Investigating stdout results from 'too_verbose' method ")
  msg = mystdout.getvalue()
  print(msg)
```

Exécution

```
Start running task ...
End running task
Investigating stdout results from 'too_verbose' method
0
1
2...
```

Exercise 1.1
prévisionnel de 1h

Rappels > E/S sur fichier

- La fonction `open` , l'objet `file`
  - `open(filename, mode)`
    ouvre un fichier de nom filename en mode lecture, écriture ou ajout (`mode="r"` ou `"w"` ou `"a"`)
    renvoie un objet de type `file` qui est un objet itérable
    `open` peut s'utiliser avec `with` sous la forme `with open('workfile', 'r') as afile :`
    l'avantage est que le fichier est automatiquement fermé, même en cas d'exception
  - `file`
    propose les méthodes `read(size-in-bytes)`, `write(string)`, `close()` et les données `name` et `mode`
    propose un `readline()` qui lit une ligne avec la fin de ligne (déjà vu)
    ne propose pas de `writeline()`, utiliser `write()` qui prends une chaine de caractères en paramètre (déjà vu)
- Robustesse du code
  - Les accès aux fichiers (comme aux ressources en général) doivent être vérifiés
    Beaucoup de méthodes comme par exemple `open()` renvoi une exception sur cas d'erreur
    Il faut utiliser le mécanisme des gestions ad-hoc via `try ... except`
  - Encodage des caractères
    Avec les accès aux fichiers se pose la question de l'encodage des chaines de caractères
    Pour plus de précision voir le complément en fin de présentation
- Autres
  - module `tempfile`, avec par exemple les méthodes `mkdtemp()`, `mkstemp()`
  - module `os` avec par exemple les méthodes
    `os.unlink()` pour supprimer un fichier
    `os.getcwd()`
    `os.walk(...)` parcours de répertoire
    `os.mkdir(path[, mode])` pour créer un répertoire

- Ecriture puis relecture d'un fichier texte

Code

```
import sys
import os
import tempfile

filename="" # access possible as global variable using 'global'
def write_file():
  global filename
  fileno,  filename = tempfile.mkstemp(".txt", dir="/tmp")  # will create a /tmp/tmpatkvzk4h.txt
  afile = open(filename, "w");
  print("Openning file %s in mode %s\n" %(afile.name, afile.mode))
  for i in range(1, 4):
    line = "%s%d\n" %("line-", i) # adding CRLF in string line
    afile.write(line)
  afile.close()

def read_file():
  afile = open(filename, "r");
  # with open(filename, "r") as afile :   # alternative syntax
  print("Openning file %s in mode %s\n" %(afile.name, afile.mode))
  # caution line include CRLF (strip should be used)
  for line in afile:
    print(line.rstrip())
  afile.close() # close useless if using 'with open '

def remove_file():
  print("Removing file %s \n" %(filename))
  os.unlink(filename)
```

Exécution

```
Openning file /tmp/tmpatkvzk4h.txt in mode w
Openning file /tmp/tmpatkvzk4h.txt in mode r
line-1
line-2
line-3
Removing file /tmp/tmpatkvzk4h.txt
```

- Module `csv`
  - Classe `reader`
    Crée à partir d'un fichier ouvert en lecture
    Fourni un itérateur sur les lignes , puis les cellules d'une ligne
  - Classe `sniffer`
    Pour déterminer si le fichier a un header
  - Classe `writer`
    Crée à partir d'un fichier ouvert en écriture
    Méthode `writerow()` pour écrire au format csv, la liste passée en argument

    Ces 3 classes sont celles utilisées dans l'exemple suivant
  - Classes `csv.DictReader` et `csv.DictWriter`
    Il est possible d'addresser un fichier CSV en le considérant comme un dictionnaire (les clés étant fournies en paramètre du constructeur)

- Processing d'un fichier CSV pour ajouter une colonne

Code

```
import csv
...

def process_csv_file():
  INPUT_FILENAME = "temperatures.csv"
  # test if file has header using Sniffer
  input_file = open(INPUT_FILENAME, "r")
  sniffer = csv.Sniffer()

  has_header = sniffer.has_header(input_file.read(1024))
  input_file.seek(0)  # point to beginning of file(after sniffer)
  # create a reader
  csv_reader = csv.reader(input_file)
  if (has_header):
    header = next(csv_reader)

  OUTPUT_FILENAME = "temperatures2.csv"
  output_file = open(OUTPUT_FILENAME, "w")
  # create a writer
  csv_writer = csv.writer(output_file)
  if (has_header):
    new_header = list(header)
    new_header.append("temperature_F")
    csv_writer.writerow(new_header)

  # T(°F) = T(°C) ? 9/5 + 32
  for row in csv_reader:
    new_row = list(row)
    new_row.append((float(row[1]) * 9 / 5) + 32)
    csv_writer.writerow(new_row)

  output_file.close()
  input_file.close()
```

Fichiers d'entrée et fichier généré

```
> more temperatures.csv
town,   temperature_C
Paris, 10
Madrid, 20
Oslo, 8


> more temperatures2.csv
town,   temperature_C,temperature_F
Paris, 10, 50.0
Madrid, 20, 68.0
Oslo, 8, 46.4
```

- Composants
  - `Logger`
    La classe qui fournie les méthodes telles que `debug()`, `info()`, `warn()`, `error()`, `critical()`
    `setLevel()` pour définir le niveau de seuil bas (rejeté si en deçà)
    (niveaux de log : `DEBUG < INFO < WARNING < ERROR < CRITICAL`)
    `addHandler()` pour associer un (ou plusieurs) handler(s)
  - `Handler`
    Classe qui permet d'adresser la destination, typiquement un fichier
    il existe de nombreux Handlers tels que `FileHandler`, `RotatingFileHandler`, `TimedRotatingFileHandler`, `SysLogHandler`, `HTTPHandler` , etc.
    ex : `TimedRotatingFileHandler(filename, when="midnight", backupCount=7)`
    `setLevel()` pour définir le niveau de seuil bas (rejeté si en deçà)
    Le handler peut redéfinir le niveau de log qu'il accepte ou rejette (en cohérence avec le niveau du Logger)
    `setFormatter()` pour associer un `Formatter`
    Détails Logging handlers
  - `Formatter`
    Pour définir les éléments à logger, en utilisant des attributs
    Ex : `MY_FORMAT = "%(asctime)-24s %(process)d %(levelname)-6s %(message)s"`
    Détails LogRecord attributes
  - `Filter`
    Pour un filtrage complémentaire au niveau de sévérité (par exemple sur le contenu)

- Exemple utilisant un logger et deux handlers

Code

```
from os.path import expanduser
import logging
from logging.handlers import TimedRotatingFileHandler
...
def do_log():
  MY_FORMAT = "%(asctime)-24s %(process)d %(levelname)-6s %(message)s"

  # Logger
  my_logger = logging.getLogger("Demo")
  my_logger.setLevel(logging.DEBUG)  # lowest level, all msg logged

  #  Handlers
  filename = os.path.expanduser('~') + "/exercises.log"
  logfileHandler = TimedRotatingFileHandler(filename, when="H", backupCount=7)
  logfileHandler.setLevel(logging.DEBUG)  # lowest level, all msg logged

  stderrHandler = logging.StreamHandler(sys.stderr)
  stderrHandler.setLevel(logging.ERROR)  # only error and above

  # Formatter
  logFormatter = logging.Formatter(MY_FORMAT)

  # add Formatter to Handlers
  logfileHandler.setFormatter(logFormatter)
  stderrHandler.setFormatter(logFormatter)

  # add Handlers to logger
  my_logger.addHandler(logfileHandler)
  my_logger.addHandler(stderrHandler)
  ...
```

```
...
# let's run computing task
my_logger.info("Start computing values")
for i in range(0, 3, 1):
  my_logger.debug("Iteration # {}".format(i))
if (True):
  my_logger.error("An error occurred, (don't worry it's a fake
my_logger.info("End computing values")
```

- Exécution du code précédent
  - Sortie sur stderr

```
2017-10-19 13:57:37,635  81956 ERROR  An error occurred, (don't worry it's a fake !)
```

  - Sortie dans le fichier $HOME/exercises.log

```
2017-10-19 13:57:37,634  81956 INFO   Start computing values
2017-10-19 13:57:37,635  81956 DEBUG  Iteration # 0
2017-10-19 13:57:37,635  81956 DEBUG  Iteration # 1
2017-10-19 13:57:37,635  81956 DEBUG  Iteration # 2
2017-10-19 13:57:37,635  81956 ERROR  An error occurred, (don't worry it's a fake !)
2017-10-19 13:57:37,635  81956 INFO   End computing values
```

- Paramétrage par fichier de config
  Il est naturellement possible de tout définir par configuration (plutôt que dans le code)
  Il existe 2 formats de fichier `configparser` et `dictConfig`

  - `configparser`
    Un fichier de configuration composé de sections : [loggers], [handlers] et [formatters]
    Voir ex en page suivante
    Lecture et utilisation du fichier :

```
filename =  "./logger4exercises.conf"
logging.config.fileConfig(filename)

my_logger = logging.getLogger()
```

  - `dictConfig`
    Un fichier de configuration au format YAML
    Lecture du fichier .yml et utilisation :

```
import yaml
...
filename =  "./logger4exercises.yml"
file  = open(filename, 'r')
conf_as_dict = yaml.load(file)
logging.config.dictConfig(conf_as_dict)

my_logger = logging.getLogger()
```

- Fichier de configuration 'logger4exercises.conf' au format `configparser`

```
[loggers]
keys=root

[handlers]
keys=logfile_handler,stderr_handler

[formatters]
keys=my_formatter

[logger_root]
level=DEBUG
handlers=logfile_handler,stderr_handler

[handler_logfile_handler]
class=handlers.TimedRotatingFileHandler
level=DEBUG
formatter=my_formatter
when=midnight
backupCount=7
args=('/tmp/exercises.log',)

[handler_stderr_handler]
class=StreamHandler
level=ERROR
formatter=my_formatter
args=(sys.stderr,)

[formatter_my_formatter]
format=%(asctime)-24s %(process)d %(levelname)-6s %(message)s
```

- Fichier de configuration 'logger4exercises.yml' au format yml pour utilisation par `dictConfig`

```
version: 1  # version is mandatory
formatters:
  my_formater:
    format: '%(asctime)-24s %(process)d %(levelname)-6s %(message)s'
handlers:
  stderr_handler:
    class: logging.StreamHandler
    level: ERROR
    formatter: my_formater
    stream: ext://sys.stderr
  logfile_handler:
    class : logging.handlers.TimedRotatingFileHandler
    level: DEBUG
    filename : '/tmp/exercises.log'
    when : midnight
    backupCount : 7
    formatter: my_formater

# root keyword is mandatory
root:
  level: DEBUG
  handlers: [logfile_handler, stderr_handler]
```

- Logging facile
  Il est possible de disposer d'un logging minimal via la méthode `basicConfig` et dont la destination est `stderr`.
  Possible de spécifier `stdout` via le paramètre `stream` ainsi `logging.basicConfig(...,stream=sys.stdout)`
- Exemple

Code

```
import logging

MY_FORMAT = "%(asctime)-24s %(levelname)-6s %(message)s"
logging.basicConfig(format=MY_FORMAT, level=logging.INFO)

my_logger=logging.getLogger()

my_logger.info("I have something to tell you")
```

Exécution

```
2017-05-31 18:16:19,380  INFO   I have something to tell you
```

Exercise 1.2
prévisionnel de 1h

- Python 2.7 sans aucune directive : Type `str`, encodage `ascii`
  - type `str` : Type par défaut d'une séquence de caractères écrit sous la forme `s = "hello"`
  - `ascii` : l'encodage par défaut (limité en possibilité de représentation de caractères)
  - Exemple

```
#!/usr/bin/env python

from platform import python_version
import sys

print("Python version ", python_version())
print("sys.getdefaultencoding() ", sys.getdefaultencoding())

s="Hello"
print("type(s) ", type(s))
```

```
('Python version ', '2.7.10')
('sys.getdefaultencoding() ', 'ascii')
('type(s) ', <type 'str'>)
```

Mais avec l'ascii, on ne peut pas spécifier des caractères accentués du Français par exemple

- Python 2.7 avec directive : Type `unicode` et encodage du fichier en `utf-8`
  - type `unicode` : Type pour spécifier des caractères autres que ascii (ex `s = "Cédric"`)
    > Obtenu en préfixant la chaine par la lettre `u` , comme ceci `u"Cédric"`
    ou
    > via la directive `from __future__ import unicode_literals` (à mettre avant tout autre import)
  - directive `# -*- coding: utf-8 -*-` : pour que l'interpréteur Python accepte les caractères autres que ascii, y compris dans les commentaires
  - Exemple

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from platform import python_version
import sys

print("Python version ", python_version())
print("sys.getdefaultencoding() ", sys.getdefaultencoding())

s1="Cédric"
print("type(s1) ", type(s1), len(s1))

s2=u"Cédric"
print("type(s2) ", type(s2), len(s2))
```

```
('Python version ', '2.7.10')
('sys.getdefaultencoding() ', 'ascii')
('type(s1) ', <type 'str'>, 7)          # 7 ?!
('type(s2) ', <type 'unicode'>, 6)
```

A noter le comportement de la méthode `len(...)`

- Et avec Python 3
  - type `str` : Type des chaines de caractères et pouvant contenir des caractères autres que ascii
    Le type `unicode` de Python 2 est devenu le type `str` en Python 3
    (le type `str` de Python 2 devient le type `bytes` en Python 3)
  - `utf-8` est l'encodage par défaut
  - Aucune directive n'est nécessaire
  - Exemple

```
#!/usr/bin/env python

from platform import python_version
import sys

print("Python version ", python_version())
print("sys.getdefaultencoding() ", sys.getdefaultencoding())

s1="Cédric"
print("type(s1) ", type(s1), len(s1))

s2=u"Cédric"
print("type(s2) ", type(s2), len(s2))
```

```
Python version  3.6.1
sys.getdefaultencoding()  utf-8
type(s1)  <class 'str'> 6
type(s2)  <class 'str'> 6
```

- `encode(...)`, `decode(...)`

  Méthodes qui vont permettre d'appliquer un encodage sur une chaine / un décodage sur une séquence de caractères

  - `encode(...)`

    Pour encoder une chaine (dans une représentation gérable par la machine (ex stockage en fichier))
    L'objet encodé à un type séquence de bytes (de type `str` avec Python 2, `bytes` avec Python 3)

  - `decode(...)`

    Pour décoder une séquence de bytes en une chaine
    L'objet décodé à un type `unicode` avec Python 2.7, `str` avec Python 3

- Exemple avec Python 2.7

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

from platform import python_version
import sys

print("Python version ", python_version())
print("sys.getdefaultencoding() ", sys.getdefaultencoding())

my_string = "Bernard"
my_string_encoded = my_string.encode()
my_string_encoded_decoded = my_string_encoded.decode()

print("type(my_string) ", type(my_string), len(my_string))
print("type(my_string_encoded) ", type(my_string_encoded), len(my_string_encoded))
print("type(my_string_encoded_decoded) ", type(my_string_encoded_decoded), len(my_string))
```

```
(u'Python version ', '2.7.10')
(u'sys.getdefaultencoding() ', 'ascii')

(u'type(my_string) ', <type 'unicode'>, 7)
(u'type(my_string_encoded) ', <type 'str'>, 7)
(u'type(my_string_encoded_decoded) ', <type 'unicode'>, 7)
```

- Exemple avec Python 3

```
#!/usr/bin/env python


from platform import python_version
import sys

print("Python version ", python_version())
print("sys.getdefaultencoding() ", sys.getdefaultencoding())


my_string = "Cédric"
my_string_encoded = my_string.encode()
my_string_encoded_decoded = my_string_encoded.decode()


print("type(my_string) ", type(my_string), len(my_string))
print("type(my_string_encoded) ", type(my_string_encoded), len(my_string_encoded))
print("type(my_string_encoded_decoded) ", type(my_string_encoded_decoded), len(my_string))
```

```
Python version  3.6.1
sys.getdefaultencoding()  utf-8

type(my_string)  <class 'str'> 6
type(my_string_encoded)  <class 'bytes'> 7
type(my_string_encoded_decoded)  <class 'str'> 6
```

- A noter
  - Aucune directive n'est nécessaire
  - Remarquer les types `str` et `bytes`

- Conclusion
  - Avec Python 2 : Complexe
    > Directives `# -*- coding: utf-8 -*-` et `from __future__ import unicode_literals` à mettre systématiquement
    > L'encodage par défaut est ascii, or les accès aux ressources (par exemple les fichiers) doivent préciser l'encodage `utf-8`
    Pour cela il faut utiliser la méthode open de la classe `codecs` (par exemple `codecs.open(..., encoding='utf-8')`)
    La fonction `open(...)` n'utilise que l'encodage par défaut.
  - Avec Python 3 : La vie est plus facile
    Encodage par défaut est `utf-8` , une chaine est de type `str` , la version sérialisée est de type `bytes`

    Pour les accès au fichier la fonction `open(..)` dispose du paramètre `encoding`
    Si rien n'est précisé l'encodage utilisé est très probablement `UTF-8`
    (consulter `locale.getpreferredencoding()`)
    La fonction `open` permets de préciser l'encodage par `open(..., encoding='utf-8')`

- Lecture / écriture de fichiers, en utf-8, avec Python 2.7

Code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

import tempfile

# With Python 2, default encoding is ascii, we must  use codecs
# e.g. codecs.open(... encoding='utf-8')  to process files in utf-8
import codecs

filename = ""
def write_file():
  global filename
  fileno, filename = tempfile.mkstemp(".txt", dir="/tmp")
  afile = codecs.open(filename, "w", encoding='utf-8');
  print("Openning file %s in mode %s" % (afile.name, afile.mode))

  names = ["Cédric", "François", "Torbjörn"]
  for name in names:
    line = "{}\n".format(name)
    afile.write(line)
  afile.close()

def read_file():
  global filename
  afile = codecs.open(filename, "r", encoding='utf-8');
  print("Openning file %s in mode %s" % (afile.name, afile.mode))
  for line in afile:
    print(line.rstrip())
  afile.close()
```

Exécution

```
Openning file /tmp/tmpjzlMOF.txt in mode wb
Openning file /tmp/tmpjzlMOF.txt in mode rb
Cédric
François
Torbjörn
Removing file /tmp/tmpjzlMOF.txt
```

- Lecture / écriture de fichiers, en utf-8, avec Python 3

Code

```
#!/usr/bin/env python

import tempfile

# With Python 3, default encoding is utf-8, cool !

filename=""
def write_file():
  global filename
  fileno,  filename = tempfile.mkstemp(".txt", dir="/tmp")
  afile = open(filename, "w" , encoding='utf-8')
  print("Openning file %s in mode %s" % (afile.name, afile.mode))

  names=["Cédric", "François", "Torbjörn"]
  for name in names:
    line = "{}\n".format(name)
    afile.write(line)
  afile.close()

def read_file():
  global filename
  afile = open(filename, "r")
  print("Openning file %s in mode %s" % (afile.name, afile.mode))
  for line in afile:
    print(line.rstrip())

  afile.close()
```

Exécution

```
Openning file /tmp/tmptml2uc4c.txt in mode w
Openning file /tmp/tmptml2uc4c.txt in mode r
Cédric
François
Torbjörn
Removing file /tmp/tmptml2uc4c.txt
```

- Point sur les directives `from __future__ import ...` dans Python 2
  Pour forcer, en utilisant Python 2 , le comportement qu'on aurait avec Python 3
  - `from __future__ import division`
    Par défaut : `1/2` donne `0`
    Avec la directive : `1/2` donne `0.5`
  - `from __future__ import print_function`
    Par défaut : `print "hello"`
    Avec la directive : ~~`print "hello"`~~ interdit, utiliser `print("hello")`
  - `from __future__ import unicode_literals`
    Par défaut : une chaine de caractère est `str`
    Avec la directive : une chaine de caractère est `unicode`, on peut écrire `name = "François Müller"`
  - `from __future__ import absolute_import`
    à voir ...

  Liste non exhaustive, pour plus de détails voir __future__ de Python 2.7

- Vue rapide
  - Module
    fichier contenant des instructions python et des fonctions et/ou classes. peut être exécuté ou importé dans un autre module
    Les instructions (autres que fonctions et classes) sont aussi exécutées dans le cas d'un import (= initialisation du module)
  - Package
    C'est un répertoire contenant des fichiers .py (donc des modules) + le fichier `__init__.py`
    `__init__.py` même vide est un marqueur obligatoire pour faire d'un répertoire un package mais ce fichier permet aussi de spécifier les règles d'imports. Voir la doc pour détails

    Attention : Dans notebook, le mécanisme d'import de fichier au format notebook (.ipynb) est bien plus complexe. Voir Importing Jupyter Notebooks as Modules
- Exemple (hors notebook)

```
>ls  pck1/
m2.py,  m1.py, __init__.py

>cat pck1/m1.py
#!/usr/bin/env python
msg="Hello you"

>cat pck1/m2.py
#!/usr/bin/env python
msg="Bye bye !"

>cat mpd.py
#!/usr/bin/env python

from pck1 import m1
from pck1 import m2
print(m1.msg, " &", m2.msg)
```

Exécution

```
# Execution de mpd.py, avec presence de pck1/__init__.py
>mpd.py
Hello you & Bye bye !

# Execution de mpd.py, après suppression de pck1/__init__.py
>\rm pck1/__init__.py*

>mpd.py

ImportError: No module named pck1
```