

Objectifs :

- Apprivoiser le (langage) **Python**.
- Réviser les concepts de programmation *système/réseau* :
 - Communication entre processus,
 - Signaux, primitives *fork*, *pipe* et *exec*, sockets
- Savoir utiliser le module *subprocess*



Documentation officielle Python v3.7 sur le Web :

<http://docs.python.org/3.7/tutorial>

<http://docs.python.org/3.7/library>

Un bon tutoriel sur Python (V3) : *moodle*/section 4 → Une introduction à Python 3 (au format pdf) de Bob Cordeau et Laurent Pointal.

Documentation en mode « ligne de commande » : *pydoc* (sans argument, la commande affiche la liste de ses options)

Partie I : Manipulation de fichiers et de commandes (Unix) en Python

Les programmes présentés ci-après sont disponibles sur *moodle*.

I.1. Exemples de traitement de **fichiers** (texte) : à comprendre et tester

i `sys.argv` tableau/liste des paramètres d'un script (nom du script inclus)
`sys.argv[0]` le chemin d'accès du script, comme `$0` en *shell*
`sys.argv[1:]` tranche de la liste de 1 jusqu'à la fin ; équivalent à `sys.argv[1:len(sys.argv)]`
 D'une manière plus générale : `liste[i:j]` est une tranche de `liste` allant de `i` (inclus) à `j` (exclu)
Remarque : on peut utiliser la notation `liste[-1]` pour accéder au dernier élément de la liste

- Compter le nombre de lignes d'un fichier de données

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
# usage: wc.py file

import os, sys          # import de modules (packages)
if len(sys.argv[1:]) == 0:
    sys.stderr.write("argument fichier manquant !\n")
    exit(1)
file = sys.argv[1]
if not os.path.isfile(file):
    msg = file + " n'est pas un fichier regulier\n"
    sys.stderr.write(msg)
    exit(1)

nbl = 0
f = open(file, 'r')      # f est un objet de type 'file' ouvert en lecture
# parcours ligne après ligne
while f.readline():      # tant qu'il y a une ligne à lire dans f
    nbl += 1              # nbl = nbl + 1
f.close()
print("nbre de lignes de", file, ":", nbl)
```

On utilise la fonction *open* pour ouvrir le fichier et la méthode *readline* pour le lire.

Rem. : Si on veut afficher les lignes du fichier, il faudra écrire :

```
ligne = f.readline()     # lecture de la 1ere ligne
while ligne:             # tant que ligne n'est pas vide
    nbl += 1
    print(ligne)
    ligne = f.readline() # lecture du suivant
```

Variante pour la partie de parcours des lignes du fichier :

```
f = open(file) # idem que open(file, 'r')
nbl = 0
for ligne in f: # un objet de type 'file' est itérable
    nbl += 1    # nbl = nbl + 1
    # print(ligne) # si on veut afficher la ligne
f.close()
print("nombre de lignes de", file, ":", nbl)
```

Itération *for...in...* pour la lecture des lignes une par une

- Version de *wc.py* avec un traitement (interception) d'exception lors de l'ouverture du fichier

```
#!/usr/bin/python3
# usage: wc1.py file

import sys
if len(sys.argv[1:]) == 0:
    sys.stderr.write("argument fichier manquant !\n")
    # idem : print("argument fichier manquant !", file=sys.stderr)
    exit(1)
file = sys.argv[1]
nbl = 0
try:
    f = open(file, 'r') # f est un objet de type file
except IOError:
    sys.stderr.write("Pb ouverture de " + file + "\n")
    # print("Pb ouverture de", file, file=sys.stderr)
else:
    # si ouverture ok
    # parcours du fichier ligne après ligne
    while f.readline():
        nbl += 1
    f.close()
print("nombre de lignes de", file, ":", nbl)
```

Interception d'une exception dans un bloc comportant les clauses *try*, *except* et *else*

- Et encore une variante de *wc.py* !

```
#!/usr/bin/python3
# -*- coding: UTF-8 -*-
# usage: wc2.py file

from sys import argv, stderr
from os.path import isfile

if len(argv[1:]) == 0:
    stderr.write('argument fichier manquant !\n') ; exit(1)
file = argv[1]
if not isfile(file):
    msg = file + " n'est pas un fichier regulier\n"
    stderr.write(msg)
    exit(1)
f = open(file, 'r')
print("nombre de lignes de", file, ":", len(f.readlines()))
f.close()
```

Ici, on importe seulement les fonctions nécessaires. Dans le script, on n'a pas besoin de préfixer celles-ci par le nom du module (sous-module)

Utilisation de *readlines()* pour lire en « un coup » **tout** le fichier
Inconvénient : peut « planter » si fichier trop volumineux !

... Je vous sens impatient de passer à l'action et d'en découdre avec le python ...



I.2. Exercice : script *joueurs_par_nation.py* *nation*

On dispose d'un fichier *foot_2020.txt* (disponible sur *moodle*) qui recense des joueurs de clubs évoluant en *Champion's League*.

Structure du fichier :

nom:club:poste:annee_naissance:nation

Exemples de valeurs de *nation* : 'fr', 'es', 'br' (Brésil)...

Valeurs possibles pour *poste* : 'g' (gardien), 'd' (défenseur), 'md' (milieu défensif), 'mo' (milieu offensif), 'a' (attaquant)



L'objectif du script est de lister, à partir du fichier, les joueurs dont la nationalité est la nation passée en paramètre (exemple : fr) ; les informations à afficher pour chaque joueur sont : son nom, le nom de son club et son âge (à calculer).

Le programme doit contrôler qu'il y a un paramètre sur la ligne de commande et que le fichier existe bien (dans le répertoire courant). D'éventuels arguments en surnombre sont purement ignorés (pas de message d'erreur).

Exemples d'exécution

```
$ ./joueurs_par_nation.py
Usage: joueurs_par_nation.py nation
$ ./joueurs_par_nation.py 2>/dev/null # pas de message d'erreur sur la sortie standard
```

```
$ ./joueurs_par_nation.py es
foot_2020.txt : Pb ouverture # se produit si 'foot_2020.txt' n'existe pas dans le dossier courant (ou s'il manque le droit de lecture)
$ ./joueurs_par_nation.py br          # lancé en 2021
Neymar, PSG, 29
Alisson, Liverpool, 29
Firmino, Liverpool, 30
Marquinhos, PSG, 27
Fernandinho, Manchester City, 36
```

Affichage plus « sophistiqué » avec alignement des colonnes à gauche

```
Neymar      | PSG              | 29 ans
Alisson     | Liverpool        | 29 ans
Firmino     | Liverpool        | 30 ans
Marquinhos  | PSG              | 27 ans
Fernandinho | Manchester City  | 36 ans
```

Indications :

- Pour la manipulation des dates (trouver l'année courante), utiliser le module `datetime`.
- Méthode `endswith(motif)` utilisable pour un objet de type `string`.
Attention au `\n` qui est récupéré en fin d'une ligne (chaîne) lue dans un fichier !
- `chaine[i:j]` est une tranche de `chaine` allant de `i` (inclus) à `j` (exclu)
`chaine[0:-1]`: sous-chaîne depuis le 1^{er} caractère jusqu'au dernier (exclu) donc jusqu'à l'avant-dernier
`chaine[:-1]`: idem
- Méthode `split('car')` qui convertit une chaîne (`str`) en une liste de tous les « mots » de la chaîne, en utilisant le caractère `car` comme séparateur (par défaut, les espaces et tabulations).

Exemples :

```
chaine = "Lloris:Spurs Tottenham:g:1986:fr"
liste = chaine.split(':')
print(liste)
['Lloris', 'Spurs Tottenham', 'g', '1986', 'fr']
print(liste[0])
Lloris

print(liste[-1])          # liste[-1] : le dernier élément ; idem que liste[len(liste) - 1]
fr

nom, club, poste, annee, nation = chaine.split(':') # plusieurs variables pour « recevoir » le résultat
print(club)
Spurs Tottenham

On peut utiliser _ (variable anonyme) si l'on ne s'intéresse pas à la valeur d'un élément :
nom, club, _, _, nation = chaine.split(':')

On peut utiliser la notation *variable pour capturer dans variable le reste de la liste :
nom, club, *reste = chaine.split(':')
print(reste)
['g', '1986', 'fr']
print(*reste)
g 1986 fr
```

- La fonction `print()` et des exemples de chaîne formatée :

```
- avec la méthode format :                                     # conseillé en python 3
i = 1 ; j = 2
s = 'i vaut {0:d} et j vaut {1:d}'.format(i, j)
print(s)
i vaut 1 et j vaut 2
s = "i vaut {1:d} et j vaut {0:d}".format(j, i) # idem que l'affectation précédente

- variante, notation %(...) :
s = 'i vaut %d et j vaut %d' % (i, j)          # ok en python 2 et 3 (mais sera obsolète un jour ...)
print(s)
i vaut 1 et j vaut 2
print('i vaut %d et j vaut %d' % (i, j))
i vaut 1 et j vaut 2
```

I.3. Un exemple de traitement du résultat renvoyé par une commande Unix

```
#!/usr/bin/python3
# ls_subprocess1.py
''' Affichage des noms des fichiers réguliers et du nombre de fichiers du repertoire courant '''
# Principe: communication/synchronisation par "pipe" entre la commande 'ls' et le script python

import os, subprocess
```

```
# Lancement d'une commande unix qui s'exécute dans un sous-processus
# l'argument stdout=subprocess.PIPE dans Popen est utile pour que le script
# puisse capturer les résultats (la sortie standard) de ce sous-processus
```

On utilise la fonction *Popen* du module *subprocess*

```
pipe = subprocess.Popen('ls', '-a', stdout=subprocess.PIPE)
```

```
# variante: pipe = subprocess.Popen('ls -a', shell=True, stdout=subprocess.PIPE)
```

```
nbf = 0
ligne = pipe.stdout.readline().decode() # 1ère lecture avant la boucle
while ligne: # tant que ligne non vide
    ligne = ligne[:-1] # on se débarrasse du \n de fin de ligne
    if os.path.isfile(ligne):
        print(ligne)
        nbf += 1
    ligne = pipe.stdout.readline().decode() # lecture ligne suivante
pipe.stdout.close()
print("Nombre de fichiers ordinaires: ", nbf)
```

Rem: `ligne[:-1]` c'est pareil que `ligne[0:-1]`, c.a.d. la tranche de ligne de 0 au dernier (exclu !)

Lecture des lignes de résultat avec *readline*

```
# variante avec boucle for ... in pour le parcours des lignes de résultat
for ligne in pipe.stdout:
    ligne = ligne[:-1] # on se débarrasse du \n de fin de ligne
    if os.path.isfile(ligne):
        print(ligne.decode())
    nbf += 1
```

Itération *for ... in ...* pour la lecture des lignes de résultat

Explication complémentaire :

La commande lancée par `subprocess.Popen` (en l'occurrence `ls -a`) est exécutée dans un **sous-processus**, fils du processus (python) courant, et en **arrière-plan**. Donc après cette instruction, le programme python continue son déroulement normalement pendant que la commande s'exécute.

La synchronisation est réalisée par l'instruction `pipe.stdout.readline()` qui est **bloquante** s'il n'y a encore aucune ligne à « consommer ».

Rem. : Par défaut, `Popen` renvoie les lignes de résultat en mode "byte string" d'où l'emploi de la fonction `decode` après `readline`, qui fait la conversion *byte string* → *str*

Regardez sur *moodle* la variante `ls_subprocess11.py` qui utilise le paramètre `universal_newlines=True` dans `Popen` pour changer ce comportement par défaut.

I.4. Exercice : scripts `joueurs_nation_parent.py` et `joueurs_nation_subproc.py`



On reprend l'exercice vu en I.2. (relire l'énoncé si vous avez la mémoire courte ...).

Cette fois-ci, vous devez écrire deux scripts qui collaborent pour réaliser le traitement voulu :

- `joueurs_nation_subproc.py`, qui est un programme de type **producteur**
objectif : filtrer les footballeurs dont la nationalité est la nation passée en paramètre et les transmettre au script `joueurs_nation_parent.py`.
- `joueurs_nation_parent.py`, qui est un programme de type **consommateur**
finalité : récupérer/lire les résultats renvoyés par `joueurs_nation_subproc.py`, afficher le nom de chaque joueur sélectionné, le nom de son club et l'âge (à calculer).

Exemples d'exécution :

```
$ ./joueurs_nation_parent.py
Usage: joueurs_nation_parent.py nation
$ ./joueurs_nation_parent.py fr
foot_2020.txt : Pb ouverture # si 'foot_2020.txt' n'existe pas dans le dossier courant (ou s'il manque le droit de lecture)
$ ./joueurs_nation_parent.py fr # lancé en 2021
Lloris, Spurs Tottenham, 35
Benzema, Real Madrid, 34
Varane, Real Madrid, 28
Pogba, Manchester United, 28
Kante, Chelsea, 30
```

Rem. : le script `joueurs_nation_subproc.py` doit pouvoir être appelé de manière indépendante (pour le tester)

```
$ ./joueurs_nation_subproc.py fr
Lloris:Spurs Tottenham:g:1986:fr
Benzema:Real Madrid:a:1987:fr
...
$ ./joueurs_nation_subproc.py
Usage: joueurs_nation_subproc.py nation
$ ./joueurs_nation_subproc.py fr
foot_2020.txt : Pb ouverture # si 'foot_2020.txt' n'existe pas dans le dossier courant (ou s'il manque le droit de lecture)
```

II.1. Un exemple de détournement de signaux (chronomètre) : chrono.py

```
#!/usr/bin/python3
# chrono.py
#
# création d'un processus chrono qui compte les secondes
# sur le signal SIGINT (CTRL C) le processus affiche la valeur du compteur
# sur le signal SIGQUIT (CTRL \) le processus affiche la valeur du compteur et s'arrête.
#
# tester aussi le comportement du pgme en le lançant dans un terminal
# et en envoyant des signaux depuis un autre : kill -2 pid pour déclencher SIGINT
#                                           kill -3 pid pour SIGQUIT

import os
from signal import signal, alarm, pause, SIGINT, SIGQUIT, SIGALRM

def seconde(signal, frame) :
    global nsec # pour signifier que c'est la variable nsec du "main"
                # sinon: variable locale et modif valeur non visible dans le main !
    nsec += 1
    alarm(1)    # on repositionne l'évènement SIGALRM

def inter(signal, frame) : # affiche la valeur du compteur
    print("CHRONO (", PID, ") ->", nsec, "secondes")

def arret(signal, frame) : # affiche la valeur du compteur et quitte
    print(nsec, "secondes écoulées")
    print("Fin du chronometre (", PID, ")")
    exit(0)

# main
nsec = 0 # nbre de secondes
PID = os.getpid() # le pid du (programme) processus courant
signal(SIGALRM, seconde) # on associe la fct "seconde" au signal
                        # SIGALRM(14)
signal(SIGINT, inter)   # le signal SIGINT (2) déclenche "inter"
signal(SIGQUIT, arret)  # SIGQUIT (3) lance "arret"
alarm(1) # on déclenche l'alarme dans 1 seconde
print("CHRONO (", PID, ") démarre !") # on affiche le pid
while True :
    pause() # en attente réception signal
```

Les gestionnaires
d'évènements (handlers)

Association de fonctions
(handlers) aux signaux

Boucle infinie d'attente
d'évènements

Primitives *fork* (duplication de processus) et *exec* (recouvrement) : multi-chronomètre

```
#!/usr/bin/python3
# multiChrono.py
# -*- coding: UTF-8 -*-

# MULTI chronomètre qui crée un nouveau (process) chrono à chaque réception du signal SIGTERM (15)
# A réception du signal SIGINT, le processus envoie ce signal à tous ses fils qui affichent
# chacun la valeur de leur compteur
# A réception du signal SIGQUIT le processus envoie ce signal à tous ses fils, qui affichent la
# valeur de leur compteur et s'arrêtent.

import signal, os, sys
# gestionnaires d'évènements
def nouveau(signal, frame) :
    # création d'un processus "chrono"
    try: pid = os.fork()
    except OSError: print('Fork: erreur !', file =sys.stderr)
    else: # fork OK
        if pid == 0 :
            os.execl('./chrono.py', "")
            exit(255) # seulement fait si execl échoue
        else :
            fils.append(pid) # fils += pid

            # on ajoute le PID du fils "chrono"
            print('*****')
            print('creation chrono de numero (', pid, ')')
            print('liste des fils', fils)

def inter(signal, frame) :
    # affiche la valeur du compteur de chaque chrono
```

Handler pour traiter le signal SIGTERM
qui doit créer un nouveau chronomètre

Ah enfin un fork !

Le fils exécute le code de *chrono.py* qui
"recouvre" (écrase) celui de son père ;
bref : le fils devient un chronomètre ;-)
Rappel : il faut le **droit x** sur *chrono.py* !

Le père mémorise le PID du
nouveau fils dans une liste

```

print('\nTransmission du signal SIGINT a tous les chronometres  fils \n')
for pid in fils :
    os.kill(pid, signal.SIGINT)

def arret(signum, frame) :
    print('\nTransmission du signal SIGQUIT a tous les chronometres  fils \n')
    for pid in fils :
        os.kill(pid, signal.SIGQUIT)
    exit(0)

# main
signal.signal(signal.SIGTERM, nouveau)      # association signaux/handlers
signal.signal(signal.SIGINT, inter)
signal.signal(signal.SIGQUIT, arret)
monPID = os.getpid()
fils = []      # pour mémoriser les PID des chronos
print("Je suis le gestionnaire de CHRONOMETRES !")
print("_____")
print("kill -2", monPID, "(dans autre terminal) pour affichage de tous les chronos")
print("kill -3", monPID, "(dans autre terminal) pour quitter et stop chronometres")
print("kill -15", monPID, "(dans autre terminal) pour creer un chrono")
print("_____")

# boucle attente évènements
while True : signal.pause() # en pause tant que pas de signal !

```

Communication père/fils via la fonction *pipe* qui retourne deux descripteurs : *Une chanson à boire...*



i « 99 Bottles of Beer » est une chanson traditionnelle aux États Unis et au Canada, dérivée de la version anglaise « Ten Green Bottles ». Avant de donner le code du script *bottles_of_beer.py*, voici son résultat à l'exécution, histoire de se mettre dans l'ambiance (quelque peu festive...) :

```

verse 1
99 bottles of beer on the wall,
99 bottles of beer.
Take one down and pass it around,
98 bottles of beer on the wall.
verse 2
98 bottles of beer on the wall,
98 bottles of beer.
Take one down and pass it around,
97 bottles of beer on the wall.
...
...
verse 99
1 bottles of beer on the wall,
1 bottles of beer.
Take one down and pass it around,
0 bottles of beer on the wall.
verse 100
No more bottles of beer on the wall,
no more bottles of beer.
Go to the store and buy some more,
99 bottles of beer on the wall.

verse 101
99 bottles of beer on the wall,
99 bottles of beer.
Take one down and pass it around,
98 bottles of beer on the wall.
...

```

Et ainsi de suite (ad eternam)



L'abus d'alcool nuit à la qualité de la programmation ...



Stock épuisé !

On recharge le stock et ça repart comme au début !

Code du programme à étudier/comprendre :

```

#!/usr/bin/python3
# -*- coding: utf8 -*-
# bottles_of_beer.py

# Utilise os.read(descripteur_pour_lire, nbre_d'octets_à_lire). Selon la longueur de la ligne :
#     verse = os.read(pipein, 118)
#     verse = os.read(pipein, 129)

import os
from time import sleep

```

```

def child(pipeout):
    bottles = 99
    bob = "bottles of beer" ; otw = "on the wall"
    take1 = "Take one down and pass it around"
    store = "Go to the store and buy some more"
    while True:
        if bottles > 0:
            values = (bottles, bob, otw, bottles, bob, take1, bottles - 1, bob, otw)
            verse = "%2d %s %s, \n%2d %s. \n%s, \n%2d %s %s. \n" % values
            # ici, la taille de verse est de 118 octets
            # variante avec la méthode format:
            # verse = "{0:2d} {1:s} {2:s}, \n{3:2d} {4:s}. \n{5:s}, \n{6:2d} {7:s} {8:s}. \n" \
            #         .format(bottles, bob, otw, bottles, bob, take1, bottles - 1, bob, otw)

            bottles -= 1
        else:
            bottles = 99
            values = (bob, otw, bob, store, bottles, bob, otw)
            verse = "No more %s %s, \nno more %s. \n%s, \n%2d %s %s. \n" % values
            # ici, la taille de verse est de 129 octets

            verse = str.encode(verse) # conversion string => byte object
            # variante: verse = verse.encode()
            os.write(pipeout, verse)

def parent(pipein):
    counter = 1
    while True:
        if counter % 100: # le reste de la division par 100 est différent de 0
            verse = os.read(pipein, 118)
        else: # counter est un multiple de 100
            verse = os.read(pipein, 129)
        verse = verse.decode()
        print('verse %d\n%s\n' % (counter, verse))
        sleep(0.3) # pour avoir le temps de lire :- )
        counter += 1

# main
pipein, pipeout = os.pipe()

if os.fork() == 0: # processus fils
    os.close(pipein) # fermeture du descripteur que le processus n'utilise pas
    child(pipeout)
else: # processus parent
    os.close(pipeout) # fermeture du descripteur que le processus n'utilise pas
    parent(pipein)

```

Le fils écrit dans le "pipe" via le descripteur *pipeout*

Le père lit dans le "pipe" via le descripteur *pipein*

pipein : descripteur pour lire ce qui est en sortie du "pipe"
pipeout : descripteur pour écrire (envoyer) dans le "pipe"

Normalement, la chanson s'arrête quand il n'y a plus rien à boire (couplet 99) ; ici, un couplet pour refaire le stock (!) est rajouté (cf *verse* 100) et le programme boucle à l'infini ...

Commentaires :

- Les processus père et fils communiquent via les deux descripteurs retournés par la fonction *pipe*.
- Rôle du processus **fils** producteur : produire le texte de la chanson **sans** les numéros de couplets. Les données sont envoyées au processus père par paquets de quatre lignes (correspondant à un couplet).
- Rôle du processus **parent** consommateur : récupérer les données transmises par le fils et afficher sur la sortie standard les couplets en les faisant précéder de leur numéro (*verse n*).

Je sens que l'impatience vous gagne à nouveau et que vous mourez d'envie de vous frotter au python ...



II.2. Exercice : *bottles_of_beer2.py*, une variante de *bottles_of_beer.py*

Il s'agit de reprendre le script dont on vient de voir le code.

On vous demande de réécrire le code du processus **père** pour que celui-ci effectue une lecture **ligne par ligne** des résultats envoyés par le processus fils.



« **Tuyau** » : la fonction `os.fdopen(descripteur)` qui retourne à partir d'un descripteur un objet de type *file* (pour la lecture des lignes d'un objet *file*, revoir si nécessaire les exemples de la section I.1)

Rem. : le reste du programme (programme principal et fonction *child*) est inchangé.

III.1. Exemple à tester et comprendre : echo-server.py et echo-client.py

Rem. : Le programme serveur ne traite qu'un seul client à la fois.

Code côté serveur :

```
#!/usr/bin/python3
# echo-server1.py
"""
Server side: open a TCP/IP socket on a port, listen for a message from a client, and send
an echo reply; this is a simple one-shot listen/reply conversation per client, but it goes into
an infinite loop to listen for more clients as long as this server script runs; the client may
run on a remote machine, or on same computer if it uses 'localhost' for server
"""

from socket import *                                # get socket constructor and constants
myHost = ''                                          # '' = all available interfaces on host
myPort = 50007                                      # listen on a non-reserved port number
sockobj = socket(AF_INET, SOCK_STREAM)             # make a TCP socket object
sockobj.bind((myHost, myPort))                     # bind it to server port number
sockobj.listen(5)                                   # listen, allow 5 pending connects

while True:                                         # listen until process killed
    connection, address = sockobj.accept()          # wait for next client connect
    print('Server connected by', address)           # connection is a new socket
    while True:
        data = connection.recv(1024).decode()      # read next line on client socket
        if not data: break                          # send a reply line to the client
        message = 'Echo=>' + data
        connection.send(message.encode())          # until eof when socket closed
    connection.close()
```

Variante plus structurée (d'un point de vue *algo*), sans instruction *break* dans la « boucle » interne :

```
while True:                                         # listen until process killed
    connection, address = sockobj.accept()          # wait for next client connect
    print('Server connected by', address)           # connection is a new socket
    data = connection.recv(1024).decode()          # read first line on client socket
    while data:
        message = 'Echo=>' + data                  # send a reply line to the client
        connection.send(message.encode())          # until eof when socket closed
        data = connection.recv(1024).decode()      # read next line on client socket
    connection.close()
```

Code côté client :

```
#!/usr/bin/python3
# usage: ./echo-client.py [ host [ word ... ] ]

"""
Client side: use sockets to send data to the server, and print server's reply to each message
line; 'localhost' means that the server is running on the same machine as the client, which lets
us test client and server on one machine; to test over the Internet,
run a server on a remote machine, and set serverHost or argv[1] to machine's domain name
or IP addr; Python sockets are a portable BSD socket interface, with object methods
for the standard socket calls available in the system's C library;
"""

import sys
from socket import *                                # portable socket interface plus constants
serverHost = 'localhost'                            # server name, or: 'starship.python.net'
serverPort = 50007                                  # non-reserved port used by the server

message = []
for x in ['Hello', 'network', 'world']:              # default text to send to server
    message.append(x.encode())

if len(sys.argv) > 1:
    serverHost = sys.argv[1]                         # server from cmd line arg 1
    if len(sys.argv) > 2:
        message = []                                # text from cmd line args 2..n
        for x in sys.argv[2:]:
            message.append(x.encode())
```



```

sockobj = socket(AF_INET, SOCK_STREAM)      # make a TCP/IP socket object
try:
    sockobj.connect((serverHost, serverPort)) # connect to server machine + port
except: # variante: except error:
    sys.stderr.write("Echec connexion\n")
else:
    for line in message:
        sockobj.send(line)                  # send line to server over socket
        data = sockobj.recv(1024).decode()   # receive line from server: up to 1k
        print('Client received:', data)
    sockobj.close()                         # close socket to send eof to server

```

Y en a qui portent des sockets !
Moi, je porte des lunettes ...



Application :

III.2. Exercice : wine_bottles_client.py et wine_bottles_server.py

- Le programme **client** commande un certain nombre N de bouteilles de vin sur le serveur ; N est tiré au hasard dans l'intervalle $[1,99]$.
Après envoi de sa demande, le client attend la réponse et se termine.
Le nombre de bouteilles reçues peut être inférieur à N si le stock du serveur n'est pas suffisant.
- Le **serveur** attend les requêtes de clients ; après acceptation d'une requête, il met à jour le stock puis répond au client (la quantité livrée peut être inférieure à la quantité commandée).
Il se termine lorsque le stock est épuisé.
Au départ, le stock est de 300 bouteilles.
Rem. : Prévoir un délai de 3 secondes entre la phase de réception d'une commande, son traitement, la mise à jour du stock et la phase (instruction *send*) d'envoi de la réponse (afin de simuler un traitement plus long).

➡ Voir exemple d'exécution page suivante

Comme dans l'exemple *echo-server.py* présenté précédemment, le serveur ne traite qu'une seule requête client à la fois.

Si on lance un deuxième client dans un terminal pendant que le serveur est en train d'en traiter un, ce deuxième client sera mis en attente de la fin du premier.

Nous verrons dans la série suivante un exemple de serveur « **multithreadé** », capable de traiter « en même temps » plusieurs requêtes *client*.

Exemple à l'exécution :

Le client	Le serveur
<pre>\$./wine_bottles_client.py Service non disponible \$ \$./wine_bottles_client.py 60 bouteille(s) recue(s) \$./wine_bottles_client.py 48 bouteille(s) recue(s) \$./wine_bottles_client.py 25 bouteille(s) recue(s) \$./wine_bottles_client.py 7 bouteille(s) recue(s) \$ # Plusieurs clients en « parallèle » mais servis séquentiellement \$./wine_bottles_client.py & ./wine_bottles_client.py & ./wine_bottles_client.py [1] 3187 [2] 3188 81 bouteille(s) recue(s) 75 bouteille(s) recue(s) 79 bouteille(s) recue(s) [1] Fini ./wine_bottles_client.py [2]- Fini ./wine_bottles_client.py \$./wine_bottles_client.py & ./wine_bottles_client.py & ./wine_bottles_client.py [1] 3190 [2] 3191 53 bouteille(s) recue(s) 12 bouteille(s) recue(s) Echec commande # le serveur s'est arrêté entre-temps [1] Fini ./wine_bottles_client.py [2]- Fini ./wine_bottles_client.py \$</pre>	<pre>\$./wine_bottles_server.py Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60582) Commande de 60 bouteille(s) Reste en stock : 240 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60584) Commande de 48 bouteille(s) Reste en stock : 192 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60596) Commande de 25 bouteille(s) Reste en stock : 7 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60598) Commande de 43 bouteille(s) Reste en stock : 0 bouteille(s) PLUS DE STOCK ! \$ \$./wine_bottles_server.py Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60616) Commande de 81 bouteille(s) Reste en stock : 219 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60618) Commande de 75 bouteille(s) Reste en stock : 144 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60620) Commande de 79 bouteille(s) Reste en stock : 65 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60622) Commande de 53 bouteille(s) Reste en stock : 12 bouteille(s) Serveur en attente de commande Serveur connecté par ('127.0.0.1', 60624) Commande de 33 bouteille(s) Reste en stock : 0 bouteille(s) PLUS DE STOCK ! \$</pre>

Rem. :

Si l'on n'intercepte pas l'exception (lancée par le système) dans le client qui est en attente de réponse alors que le serveur s'est arrêté entre-temps (cause stock=0), on « prend en pleine poire » des injures :

```
Traceback (most recent call last):
  File "./wine_bottles_client.py", line 19, in <module>
    reponse = socket.recv(2).decode()
ConnectionResetError: [Errno 104] Connection reset by peer
```

"That's all Folks!"