IUT Robert Schuman
Institut universitaire de technologie
Département informatique
Université de Strasbourg

S4x / Programmation concurrente (en Python)

Série 2 : *Threads*

Notions :
- Programmation concurrente :
  Section critique, verrou (*lock*), exclusion mutuelle, synchronisation
- Liste définie en compréhension (intension)
- Classe et héritage

Aspects « techniques » :
- modules *threading*, *queue*, *socket*
- objets de type *Thread*, *Lock*, *Queue*

Documentation officielle Python **v3.7** sur le Web :
http://docs.python.org/3.7/tutorial
http://docs.python.org/3.7/library

Tutoriels en ligne sur **moodle** / section 4
→ Une introduction à Python 3 (au format pdf) de Bob Cordeau et Laurent Pointal
→ *Python Concurrency* de David Beazley

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

Avant « d'attaquer » le vif du sujet, à savoir la programmation concurrente et les *threads*, voyons d'abord un programme dont l'objectif devrait vous rappeler des souvenirs.
Il s'agit pour les deux scripts *python* suivants (en ligne sur *moodle*) d'examiner, avec la commande ***ping***, chacune des machines hôtes données (en paramètre) afin de déterminer si elle est « vivante », défaillante ou inconnue (sur le réseau).

Quel manque d'imagination M. Schnell ! Vous n'avez pas trouvé mieux que cet exercice déjà traité en S11 ? ...

Rappel commande Unix `ping -w délai machineHôte`
l'option `-w` *délai* évite que *ping* ne « boucle » ; elle stoppe la commande au bout de *délai* secondes (le délai est fixé arbitrairement à 3 sec. dans le script)
autre option possible : `-c` (pour imposer le nombre de requêtes max)

Option `-t` au lieu de `-w` sous **Mac OS**/ FreeBSD !

valeur du code retour `$?` à la suite de la commande :
- 0 si OK
- 1 si pb (défaillance)
- 2 : autres cas (considérer que la machine est inconnue)

Où l'on retrouve le module *subprocess* de la série précédente ...

Dans la 1ère version ci-dessous, on utilise la primitive `call` (au lieu de `Popen` vu dans la série précédente) qui exécute la commande Unix donnée (1er paramètre), **attend** qu'elle soit finie et retourne le code de retour de la commande (c.a.d la valeur `$?`).

```
#!/usr/bin/python3
# -*- coding: utf8 -*-
# no_thread_ping_call.py  host [ host ... ]
```

```
# NB : c'est le shell qui affiche les lignes de la cde ping
#   (le script python n'intercepte pas les lignes de résultat, mais seulement le code retour)

from sys import argv, stdin, stdout, stderr
from subprocess import call, DEVNULL
from os.path import basename

if len(argv[1:]) == 0:
    stderr.write('Usage: ' + basename(argv[0]) + ' host ...\n')
    exit(1)
delai = 3  # 3 sec.
for hote in argv[1:]:
    # on lance la cde unix : ping -w délai hote
    # et on récupère ds la variable <retour> le code retour
    #    de la cde ping (cad la valeur de $? du shell)
    retour = call(['/bin/ping', '-w' + str(delai), hote])
```

Avec **call**, la commande *ping* est lancée dans un sous-processus et en **avant-plan**

```
    # ok aussi: retour = call('/bin/ping -w' + str(delai) + ' ' + hote, shell=True)
    # __ si on ne veut pas de trace du déroulement de ping, ni message d'erreur__
    # retour = call(['ping', '-w' + str(delai), hote], stdout=DEVNULL, stderr=DEVNULL)

    print('*' * 40)
    if retour == 0:
        print(hote, ': VIVANT')
    elif retour == 1:
        print(hote, ': DEFAILLANT ?')
    else:
        print(hote, ': INCONNU')
    print('*' * 40)
```

Variante (avec `Popen` au lieu de `call`)

```
#!/usr/bin/python3
# -*- coding: utf8 -*-
# no_thread_ping_Popen.py host [ host ... ]
# NB : c'est le shell qui affiche les lignes de la cde ping
#   (le script python n'intercepte pas les lignes de résultat, mais seulemt le code retour)

from sys import argv, stdin, stdout, stderr
from subprocess import Popen, DEVNULL
from os.path import basename

if len(argv[1:]) == 0:
    stderr.write('Usage: ' + basename(argv[0]) + ' host ...\n')
    exit(1)
delai = 3  # 3 sec.
for hote in argv[1:]:
    # on lance la cde unix : ping -w délai hote
    process = Popen(['/bin/ping', '-w' + str(delai), hote])

    # __ si on ne veut pas de trace du déroulement de ping, ni message d'erreur__
    # process = Popen(['/bin/ping', '-w' + str(delai), hote], stdout=DEVNULL,
    #                                                         stderr=DEVNULL)

    retour = process.wait()
    print('*' * 40)
    if retour == 0:
        print(hote, ': VIVANT')
    elif retour == 1:
        print(hote, ': DEFAILLANT ?')
    else:
        print(hote, ': INCONNU')
    print('*' * 40)
```

Avec **Popen**, la commande *ping* est lancée ds un sous-processus et en **arrière-plan** !

Attente de la fin de la cde lancée en arrière-plan et mémorisation du code retour (valeur de `$?` du shell)

Exemple d'**exécution** :

```
$ ./no_thread_ping_call.py  turlututu  google.fr  fou.iutrs.unistra.fr  taratata  localhost
ping: unknown host turlututu
****************************************
turlututu : INCONNU
****************************************
PING google.fr (193.51.224.174) 56(84) bytes of data.
64 bytes from cache.google.com (193.51.224.174): icmp_seq=1 ttl=58 time=6.94 ms
64 bytes from cache.google.com (193.51.224.174): icmp_seq=2 ttl=58 time=6.91 ms
64 bytes from cache.google.com (193.51.224.174): icmp_seq=3 ttl=58 time=7.66 ms

--- google.fr ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 6.917/7.174/7.664/0.353 ms
****************************************
google.fr : VIVANT
****************************************
PING fou.iutrs.unistra.fr (130.79.223.49) 56(84) bytes of data.

--- fou.iutrs.unistra.fr ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2015ms

****************************************
fou.iutrs.unistra.fr : DEFAILLANT ?
****************************************
ping: unknown host taratata
****************************************
taratata : INCONNU
****************************************
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.023 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.028 ms
```

Comme on peut le constater, le traitement est <u>séquentiel</u>, dans l'ordre des (machines) hôtes passés en paramètre : turlututu puis google.fr, etc

```
64 bytes from localhost (127.0.0.1): icmp_seq=4 ttl=64 time=0.026 ms

--- localhost ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.023/0.026/0.030/0.006 ms
**************************************
localhost : VIVANT
**************************************
```

Voyons le **temps** mis à l'exécution :

```
$ time ./no_thread_ping_call.py turlututu google.fr fou.iutrs.unistra.fr taratata localhost
                                                                    1>/dev/null  2>&1

real    0m9.085s
user    0m0.020s
sys     0m0.004s
```

Un peu plus de 9 secondes !

Partie I : Premiers pas avec le module threading

Un **thread** ou fil (d'exécution) ou tâche (autres appellations connues : processus léger, fil d'instruction, processus allégé, voire unité d'exécution) est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Création d'un thread : *objetThread* = threading.Thread(target=*fonction*, args=(*arg1*,*arg2*,...))
avec l'instruction d'import : `import threading`
objetThread = Thread(target=*fonction*, args=(*arg1*,*arg2*,...))
avec l'instruction d'import : `from threading import Thread, ...`

Rem. : un tuple composé d'un seul item s'écrit `(item,)` # curieux, cette virgule obligatoire...

Lancement (activation) : *objetThread*.start()

I.1. Exemple : threaded_ping.py host [host ...]

```
#!/usr/bin/python3
# -*- coding: utf8 -*-
# threaded_ping.py host ...

# NB : c'est le shell qui affiche les lignes de la cde ping
#   (le script python n'intercepte pas les lignes de résultat, mais seulemt le code retour)

from sys import argv, stdin, stdout, stderr
from subprocess import call, DEVNULL
from os.path import basename
from threading import Thread

def ping(hote, delai):
    # on lance la cde unix ping pour interroger hote
    # (option -w pour arrêter après un certain délai sinon ping "boucle")

    retour = call(['/bin/ping', '-w' + str(delai), hote])

    # Si on ne souhaite pas de trace du déroulement de ping (ni message d'erreur)
    # retour = call(['ping', '-w' + str(delai), hote], stdout=DEVNULL, stderr=DEVNULL)

    etoiles = '*' * 20

    print(etoiles)
    if retour == 0:
        print(hote, ': VIVANT')
    elif retour == 1:
        print(hote, ': DEFAILLANT ?')
    else:
        print(hote, ': INCONNU')
    print(etoiles)

# main
if len(argv[1:]) == 0:
    stderr.write('Usage: ' + basename(argv[0]) + ' host ...\n')
    exit(1)
```

Commande *ping* lancée dans un sous-processus, en avant-plan.

Pour garantir l'atomicité des *print*, il faudra protéger cette section de code par un verrou (*Lock* dans le module *threading*)

```
delai = 3
for hote in argv[1:]:
    t = Thread(target=ping, args=(hote, delai))
    t.start()
```

Exemple d'**exécution** sur un **serveur** :

```
j.schnell@troglo:~/python3/threads$ ./threaded_ping.py  turlututu  google.fr
                                                        fou.iutrs.unistra.fr  taratata  localhost
PING google.fr (193.51.224.154) 56(84) bytes of data.
PING fou.iutrs.unistra.fr (130.79.223.49) 56(84) bytes of data.
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.028 ms
ping: unknown host turlututu
ping: unknown host taratata
********************
taratata : INCONNU
********************
64 bytes from cache.google.com (193.51.224.154): icmp_seq=1 ttl=58 time=6.87 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.025 ms
64 bytes from cache.google.com (193.51.224.154): icmp_seq=2 ttl=58 time=6.88 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.027 ms
64 bytes from cache.google.com (193.51.224.154): icmp_seq=3 ttl=58 time=7.07 ms

--- google.fr ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 6.872/6.946/7.079/0.094 ms
********************
********************
turlututu : INCONNU
google.fr  : VIVANT
********************
********************

--- fou.iutrs.unistra.fr ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2015ms

********************
fou.iutrs.unistra.fr : DEFAILLANT ?
********************

--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.025/0.026/0.028/0.006 ms
********************
localhost : VIVANT
********************
```

On peut constater le traitement en « parallèle » des différents (machines) hôtes passés en paramètre.

Tiens ... un défaut à l'affichage !
On devrait avoir :

```
********************
turlututu : INCONNU
********************
********************
google.fr  : VIVANT
********************
```

Temps d'exécution :

```
j.schnell@troglo:~/python3/threads$ time ./threaded_ping.py  turlututu  google.fr
                        fou.iutrs.unistra.fr  taratata  localhost 1>/dev/null  2>&1

real    0m3.051s
user    0m0.012s
sys     0m0.020s
```

A peine plus de **3 secondes** !
A comparer avec les 9 sec. de la version sans thread

Rem. : Le défaut d'affichage constaté ci-dessus (« mauvais » enchevêtrement de lignes) peut se produire occasionnellement ; mais ce qui est sûr, c'est que le risque que ça arrive augmente si le nombre de threads augmente.
Ceci est dû à la concurrence entre threads dans la section de code des instructions *print*.

Remède : définir une **section critique** qui permet de « protéger » une zone de code afin d'y accéder en **exclusion mutuelle**.

Nous allons utiliser la technique du verrou (*mutex*) pour « encadrer » les instructions *print* et garantir ainsi que leur exécution par un thread soit indivisible (c.a.d. ne puisse pas être interrompue par un autre thread).

Caractéristiques d'un **verrou/mutex** (*Lock* en Python) :

- ✗ objet avec deux états possibles : libre (= non verrouillé) et verrouillé
- ✗ ne peut être acquis (verrouillé) que par un seul thread à la fois.
- ✗ un thread demandant à verrouiller un *Lock* qui l'est déjà est mis en attente (bloqué) jusqu'à relâchement (libération) du *Lock* (sauf cas particulier où un thread demande l'acquisition du verrou en mode *non bloquant*).
  Si plusieurs threads sont en attente d'acquisition, un seul d'entre eux pourra être satisfait lorsque le verrou sera libéré.
- ✗ une demande de relâchement d'un verrou qui est dans l'état « libre » provoque une erreur.

I.2. Faites les **modifications** nécessaires dans le script `threaded_ping.py` en tenant compte des informations/explications données précédemment.

🛠 À placer avant la définition de la fonction *ping* ou en début de *main* (n.b. : ne pas oublier d'**import**er le constructeur *Lock*)

```
mutex = Lock()    # allocation d'un verrou (état libre à sa création)
# mutex comme MUTual EXclusion
```

Je vous fais confiance (ne me décevez pas !) pour trouver les méthodes applicables à un objet de type *Lock* dont vous avez besoin dans la fonction *ping* pour encadrer la partie d'instructions d'affichage : il suffit de traduire en anglais les mots *acquérir* et *relâcher* ...

I.3. Script `threaded_ping2.py`

On reprend le script précédent et on vous demande de le modifier pour que tous les résultats finaux ('vivant', 'défaillant', 'inconnu') soient affichés par le programme principal (*main thread*), une fois que tous les hôtes ont été scrutés.

On utilise pour cela une liste partagée par tous les threads (donc une ressource critique) dans laquelle chaque thread ajoute son diagnostic sous la forme d'un couple (*host* , *code_retour*).
Il est donc nécessaire que le *main* thread **attende** la fin de tous les threads *ping* avant de parcourir la liste et afficher les résultats.

Je vous fais confiance (encore ? ...) pour trouver la méthode applicable à un objet de type *Thread* qui permet d'attendre que l'exécution de cet objet soit terminée, et donc de synchroniser le *main* avec les *threads*.
Son nom commence par la lettre j...

I.4. Script avec une **section critique** : `incremente.py`

Si jamais vous n'êtes pas convaincu de l'utilité de verrouiller une zone « sensible » d'un programme, étudiez et testez l'exemple qui suit, disponible sur *moodle* ➔ Exemples.

No ! Thread locks ...

Are these dreadlocks?

```
#! /usr/bin/python3
# Exemple pour montrer l'intérêt de verrouiller en écriture
# une ressource critique (notion de section critique)

from threading import Thread, Lock
# variables globales, communes aux 2 threads
compteur = 0
lock = Lock()

def incremente() :
    """ incremente valeur du compteur """
    global compteur
    for i in range(5000*1000):  # 5 millions
        lock.acquire()
        compteur += 1 # section critique
        lock.release()

thread1 = Thread(target=incremente)
thread2 = Thread(target=incremente)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("valeur du compteur", Compteur)
```

Variante :
```
with lock:
    compteur += 1
```

Le résultat affiché par ce script est égal à `10000000` : `2`* *la valeur de l'intervalle pour la variable* i (qui est de 5 millions)
Exécutez le programme (plusieurs fois) pour vous en assurer !

Mettez maintenant en commentaire les instructions `lock.acquire` et `lock.release` qui protègent la mise à jour du compteur
Exécutez à nouveau le programme plusieurs fois de suite ! **Que constatez-vous ?** (valeur du compteur affichée)

Et si jamais le résultat est malgré tout correct ... augmentez la valeur actuelle (5 millions) dans la fonction *range*() !
Le **problème**, si l'instruction de mise à jour n'est pas protégée, vient du fait que l'instruction `compteur += 1` n'est pas atomique !
En réalité, elle se décompose en plusieurs instructions *machine* et la commutation entre threads peut faire perdre des mises à jour.
Il faut plutôt « voir » l'instruction `compteur += 1` comme :

```
temp = compteur          # temp : variable locale au thread
incrémenter temp de 1
compteur = temp
```

En cas **d'interruption** d'un thread pendant ces instructions, **imaginez ce qu'il se passe** lorsque le thread interrompu par son concurrent « reprend la main » ...

Partie II : Exercices avec le module threading

II.1. Foot et Champion's League, le retour : `joueurs_par_nation_thread.py` *nation*

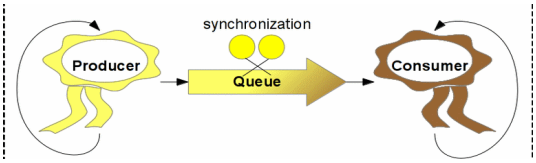
Rappel (série 1) : L'objectif est de lister, à partir d'un fichier (*foot_2020.txt*), les joueurs dont la nationalité est la nation passée en paramètre (exemple : 'fr') ; les informations à afficher pour chaque joueur sont : son nom, le nom de son club et son âge (à calculer).
Le programme doit contrôler 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).

Dans cette version, on vous demande d'utiliser **deux threads**, au lieu d'un programme/processus père (*joueurs_nation_parent.py*) et un programme/processus fils (*joueurs_nation_subproc.py*) appelé par `subprocess.Popen()` comme dans la 1ère série (Partie I.4).

Répartition des tâches entre les deux threads :

- Un thread producteur exécute le code de la fonction `writer` qui filtre les joueurs dont la nationalité est la nation passée en paramètre et les transmet au thread consommateur (toutes les colonnes étant séparées par un ':').
  Rem. : prévoir une petite pause (*sleep* ...) entre la production de deux résultats !
- Le 2ième thread, consommateur, exécute le code de la fonction `reader` qui :
  - lit les lignes transmises par le thread producteur.
  - affiche les informations : nom de joueur, nom de club et âge (à calculer).

Les deux threads communiquent via une *Queue* (structure de données de type *FIFO*).
Les fonctions `writer` et `reader` ont toutes deux comme paramètre l'objet de type *Queue*.
Rem. : les objets *Python* en paramètre d'une fonction sont transmis par **référence**.



ⓘ Les opérations de lecture/écriture dans une *Queue* sont « thread safe », c.a.d. protégées des accès concurrents (donc le programmeur n'a pas besoin de définir une section critique pour les « encadrer »).

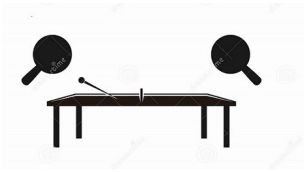
Squelette de code :

```
#!/usr/bin/python3
# -*- coding: utf8 -*-

from threading import Thread
from time import sleep
from sys import argv, stderr, stdout
from queue import Queue
from datetime import datetime
from os.path import basename

# définition de la fonction writer
def writer(fifo, ...)
    ...
    ...
# définition de la fonction reader
def reader(fifo):
    ...
    un_joueur = fifo.get()   # get bloquant   # idem que: un_joueur = fifo.get(block=True)
    while un_joueur is not None:    # None: valeur spéciale 'sentinelle' prévue par le langage
        ...
        ...
        un_joueur = fifo.get() # get bloquant
# main
# lecture/contrôles des paramètres, ouverture du fichier
...
...
# création de la file FIFO (Queue)
Fifo = Queue()
# Instanciation et démarrage des deux threads
...
# fermeture du fichier
...
```

Attention : Après avoir mis dans la *Queue* le dernier item significatif, le thread *producteur* doit y déposer une valeur spéciale « **sentinelle** » (un indicateur de fin) pour que le thread consommateur sache qu'il n'y a plus rien à lire.

II.2. Synchronisation dans une partie de ping pong : `ping_pong.py`

Il s'agit d'écrire un programme avec un certain nombre de threads (très basiques !) qui « font » (affichent) soit *ping* soit *pong*, puis s'arrêtent.
On lance autant de threads *ping* que de threads *pong* ; chaque thread *ping* fait **un** (seul) *ping* et chaque thread *pong* fait **un** (seul) *pong*.
Il s'agit ici essentiellement d'un problème de **synchronisation** (plus que de concurrence).



Exemples d'exécution :

```
$ ./ping_pong.py  3
ping ... pong
ping ... pong
ping ... pong
Fin de partie !
$ ./ping_pong.py # en l'absence de paramètre, nbre de 'ping pong' = 100
ping ... pong
ping ... pong
...
ping ... pong     # le 100ième 'ping pong'
Fin de partie !
```

Code **à compléter** :

```
#!/usr/bin/python3
# -*- coding: utf8 -*-
# ping_pong1.py
""" Un programme qui fait ping pong ping pong ... un certain nombre (N) de fois
    N threads ping et N threads pong """
# Utilisation d'un verrou (Mutex) et d'une variable d'état "bascule" (Ping_OK)
from time import sleep
from random import random
...

def ping() :
    # global Mutex   # déclaration 'global' facultative
    global Ping_OK   # déclaration indispensable car Ping_OK est m.a.j.
    sleep(random())
    ...
    ...

def pong() :
    # global Mutex   # facultatif
    global Ping_OK   # déclaration indispensable car Ping_OK est m.a.j.
    sleep(random())
    ...
    ...

# main thread
if __name__ == '__main__' :
    if sys.argv[1:] :        # si liste des paramètres non vide
        N = int(sys.argv[1])
    else :
        N = 100       # 100 ping pong par défaut
    Ping_OK = False   # pour ne pas faire pong s'il manque un ping
    Mutex = ...

    # création d'une liste de N threads ping ; chaque thread fait UN (seul) ping
    Threads_ping = [ threading.Thread(target=ping) for i in range(N) ]
```

Avertissement : pas facile ...

Va falloir se triturer les méninges !

tiens, notation originale ! Curieux, non ? voir explications ci-après (notion de **liste en compréhension**)

```
    # création d'une liste de N threads pong ; chaque thread fait UN (seul) pong
    Threads_pong = [ threading.Thread(target=pong) for i in range(N) ]
    for t in Threads_ping : t.start()
    for t in Threads_pong : t.start()
    for t in Threads_ping : t.join()
    for t in Threads_pong : t.join()
    print("\nFin de partie !")
```

Notion de **liste en compréhension**
Les listes en compréhension fournissent une façon concise de créer des listes.
Une liste en compréhension consiste en une expression suivie d'une clause `for`, puis zéro ou *plus* clauses `for` ou `if`.
Le résultat sera une liste résultant de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent.

On peut dire aussi que la liste créée de cette manière est équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.
Ainsi, l'instruction `Threads_ping = [threading.Thread(target=ping) for i in range(n)]` est équivalente à :

```
Threads_ping = []
for i in range(n) :
   Threads_ping.append(threading.Thread(target=ping) )
   # variante: Threads_ping += [ threading.Thread(target=ping) ]
```

Autres exemples : (*python* en mode interactif)

$ python3

```
>>> fruits = ['  banane', '  myrtille  ', 'fruit de la passion ']
>>> [ un_fruit.strip() for un_fruit in fruits ]  # strip supprime les espaces de début et fin de ligne (chaîne)
['banane', 'myrtille', 'fruit de la passion']    # le résultat est une liste !
>>> vec = [2, 4, 6]
>>> [ 3*x for x in vec ]
[6, 12, 18]
>>> [ 3*x for x in vec if x > 3 ]
[12, 18]
>>> valeurs = [ "10", "33", "57" ]
>>> sum( [ int(i) for i in valeurs ] )
100
```

Ce dernier exemple est équivalent à :

```
s = 0
for i in valeurs : s = s + int(i)
print(s)
```

Partie III : Daemon or not daemon ?

What the hell are you doing here, **Beastie** ?

III.1. **Daemon** thread *versus* **non daemon** thread

Où une nouvelle « race » de thread montre son visage : le thread *daemon* (thread en arrière-plan)

1ère question : Qui est Beastie ? ;-) (il y a un rapport avec la distribution B.S.D.)

Plus sérieusement :

Quels sont les **résultats** lors de l'exécution du script suivant ? (*daemon_vs_non-daemon.py*, disponible sur *moodle*)
Quelle **conclusion** en tirez-vous ? (**différence** entre les deux sortes de threads)

```
#! /usr/bin/python3
import threading, time
def daemon():
    print('daemon is starting')
    time.sleep(2)
    print('daemon is exiting')
def non_daemon():
    print('non daemon is starting')
    print('non daemon is exiting')

d = threading.Thread(target=daemon)
d.daemon = True      # d.setDaemon(True)
t = threading.Thread(target=non_daemon)
d.start()
time.sleep(0.1)      # petite temporisation
t.start()
```

Mon confrère (ci-dessus) et moi-même sommes un peu mal à l'aise d'être utilisés ici ! Un thread *daemon*, ce n'est pas vraiment comparable à un « vrai » *daemon*, je veux parler d'un processus *daemon*. Jugez sur un point par exemple :
- Un **processus** *daemon* reste en vie et continue à tourner en arrière-plan, une fois que le processus père qui l'a créé (par *fork*) « meurt » (exemple : httpd).
- Un **thread** *daemon*, lui, se termine automatiquement lorsque le programme/processus python qui l'a lancé se termine.

Ajoutez maintenant en fin de programme les instructions d'attente des deux threads, c'est à dire :

```
t.join() ; d.join()
```

Les résultats sont-ils les mêmes que précédemment ? **Expliquez**

III.2. **Bonne nuit** avec `daemon_night.py`

Un autre exemple disponible sur la plate-forme *moodle*, à étudier et à faire « tourner » pour bien comprendre comment se comporte un *daemon* thread (et le *main* thread par rapport à lui).
Histoire de bien « enfoncer le clou » : regardez aussi ce qui se passe si on supprime la ligne `t.setDaemon(True)`.

```
#! /usr/bin/python3
""" daemon TIMER  """
import threading
import time

def clock(interval):
    """ fct exécutée par le thread daemon qui fait office de timer """
    while True:
        time.sleep(interval)
        print("zzzzzzzzzz........\t\t%s" % time.ctime())

t = threading.Thread(target=clock, args=(1,))
t.setDaemon(True)    # t.daemon = True
t.start()

print("Endormissement")
for i in range(3):                  # dans ce pgme, seulement 3 cycles de sommeil pour que ca ne s'éternise pas ...
   # 1 cycle de sommeil : 90 mn en moyenne
    # 10 mn en vrai => 1 sec. dans le programme
    # 4 phases par cycle
    print("Cycle %i" %(i + 1))
    print("phase de sommeil leger") ; time.sleep(2.5)
    print("phase de sommeil lent profond") ; time.sleep(3.5)
    print("phase de sommeil paradoxal") ; time.sleep(2)
     print("phase intermediaire") ; time.sleep(1)
print("DRINGGG ! Reveil")
```

III.3. Version de *Bonne nuit* **avec** (plus de !) **classe** : `daemon_night_class.py`

```
#! /usr/bin/python3
# -*- coding: utf8 -*-
""" daemon TIMER  """

import threading
import time

class Clock(threading.Thread):
    """ thread daemon qui fait office de timer """

    def __init__(self, interval):
      threading.Thread.__init__(self) # appel au __init__
                                      # de la classe Thread
      self.daemon=True
      self.interval = interval

    def run(self):
      while True:
          time.sleep(self.interval)
          print("zzzzzzzzzz........\t\t%s" % time.ctime())

Clock(1).start()
# c = Clock(1) ; c.start()
print("Endormissement")
for i in range(3):
    print("Cycle %i" %(i + 1))
    print("phase de sommeil leger")
    time.sleep(2.5)
    print("phase de sommeil lent profond")
    time.sleep(3.5)
    print("phase de sommeil paradoxal")
    time.sleep(2)
    print("phase intermediaire")
    time.sleep(1)
print("DRINGGG ! Reveil")
```

Définition de la classe *Clock* par **héritage** de la classe *Thread*

Méthode pour initialiser les attributs de la classe **Attention** au nom `__init__` , un peu bizarre : 2 *underscore* avant et après `init` !

Méthode `run` appelée automatiquement quand on lance (par *start*) un objet d'une classe fille de la classe *Thread*

Instanciation d'un objet de la classe *Clock* et démarrage

Au lieu d'utiliser **l'héritage**, on peut aussi associer une instance d'une classe à un thread en utilisant le principe de **relation** ou de **composition** comme dans les scripts ci-dessous (disponibles sur moodle/exemples) `other_daemon_night_class.py` et `other_daemon_night_class1.py`. Jetez-y un œil, voire les deux ;-)

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

""" daemon TIMER  """
from threading import Thread
```

```
from time import sleep, ctime

class Clock:
    def __init__(self, interval):
        self.interval = interval
    def start(self):
        t = Thread(target=self.run)
        t.daemon = True
        t.start()
    def run(self):
        while True:
            sleep(self.interval)
            print("zzzzzzzzzz.........\t\t%s" % ctime())

c = Clock(1)
c.start()
print("Endormissement")
... # suite idem que version précédente
```

Variante :

```
#! /usr/bin/python3
# -*- coding: UTF-8 -*-
# other_daemon_night_class1.py

""" daemon TIMER  """
from threading import Thread
from time import sleep, ctime

class Clock:
    def __init__(self, interval):
        self.interval = interval
        self.t = Thread(target=self.run)
        self.t.daemon = True
    def start(self):
        self.t.start()
    def run(self):
        while True:
            sleep(self.interval)
            print("zzzzzzzzzz.........\t\t%s" % ctime())

c = Clock(1)
c.start() # démarre le thread associé à l'objet c
print("Endormissement")
...
```

Partie IV : Le retour des sockets avec un serveur « multithreadé » (un thread par client)

Exercice : `wine_bottles_client.py` et `wine_bottles_th_server.py`

Rappel (cf série 1) :

- 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).

Cette fois-ci, le serveur doit être capable de traiter plusieurs requêtes client « en même temps ».

Voici d'abord le code du programme client (au cas où vous n'auriez pas réussi à le coder ou pas eu le temps de le faire dans la série 1) :

```
#! /usr/bin/python3
# wine_bottles_client.py

from socket import *
from random import randint
hote = 'localhost'
port = 50007

socket = socket(AF_INET, SOCK_STREAM)
```

```
try:
    socket.connect((hote, port))
except:   # variante: except error:
    print("Serveur non disponible")
else:
    bouteilles = randint(1,99)
    demande = "{0:2d}".format(bouteilles)
    socket.send(demande.encode())
    try:
        reponse = socket.recv(2).decode()
    except:
        print("Echec commande")
    else:
        print(reponse + " bouteille(s) recue(s)")
socket.close()
```

Exercice : Écrire une version du serveur, qu'on va appeler `wine_bottles_th_server.py`, où chaque requête de client est, après acceptation (instruction *accept* dans le *main* thread), traitée dans un thread à part.

Attention : comme il y a des accès concurrents, veillez à protéger la section de code de mise à jour du stock 

Pour le code exécuté par un thread, vous avez le choix entre la définition d'une **fonction** ou d'une **classe** avec son constructeur et la méthode *run*.

Hum hum ... Pas d'indication ?
Ca m'aurait intéressé de savoir s'il y a des paramètres à passer à la fonction ou au constructeur ...

Exemple d'exécution :

Client	Serveur (on peut **constater l'entrelacement** des résultats)

`$ ./wine_bottles_client.py & ./wine_bottles_client.py`
`[2] 4275`
 	
`18 bouteille(s) recue(s)`
 	
`45 bouteille(s) recue(s)`
`[2]+  Fini  ./wine_bottles_client.py`
 	
`$ ./wine_bottles_client.py & ./wine_bottles_client.py &`
`./wine_bottles_client.py`
`[2] 4281`
`[3] 4282`
 	
`14 bouteille(s) recue(s)`
 	
`18 bouteille(s) recue(s)`
 	
`90 bouteille(s) recue(s)`
`[2]-  Fini  ./wine_bottles_client.py`
`[3]+  Fini  ./wine_bottles_client.py`
 	
`$ ./wine_bottles_client.py & ./wine_bottles_client.py &`
`./wine_bottles_client.py`
`[2] 4287`
`[3] 4288`
	`$ ./wine_bottles_th_server.py`
	`Serveur démarré`
	
	`Serveur connecté par ('127.0.0.1', 60532)`
	`Commande de 18 bouteille(s) ->`
	`('127.0.0.1', 60532)`
	
	`Serveur connecté par ('127.0.0.1', 60534)`
	`Livraison de 18 bouteille(s) ->`
	`('127.0.0.1', 60532)`
	`Reste en stock : 282 bouteille(s)`
	`Commande de 45 bouteille(s) ->`
	`('127.0.0.1', 60534)`
	`Livraison de 45 bouteille(s) ->`
	`('127.0.0.1', 60534)`
	`Reste en stock : 237 bouteille(s)`
	
	`Serveur connecté par ('127.0.0.1', 60538)`
	`Commande de 14 bouteille(s) ->`
	`('127.0.0.1', 60538)`
	
	`Serveur connecté par ('127.0.0.1', 60536)`
	`Livraison de 14 bouteille(s) ->`
	`('127.0.0.1', 60538)`
	`Reste en stock : 223 bouteille(s)`
	`Commande de 18 bouteille(s) ->`
	`('127.0.0.1', 60536)`
	
	`Serveur connecté par ('127.0.0.1', 60542)`
	`Livraison de 18 bouteille(s) ->`
	`('127.0.0.1', 60536)`
	`Reste en stock : 205 bouteille(s)`
	`Commande de 90 bouteille(s) ->`
	`('127.0.0.1', 60542)`
	`Livraison de 90 bouteille(s) ->`
	`('127.0.0.1', 60542)`
	`Reste en stock : 115 bouteille(s)`
	
	`Serveur connecté par ('127.0.0.1', 60546)`
	`Commande de 80 bouteille(s) ->`
	`('127.0.0.1', 60546)`

```
80 bouteille(s) recue(s)
```

```
Serveur connecté par ('127.0.0.1', 60544)
Livraison de 80 bouteille(s)    ->
('127.0.0.1', 60546)
Reste en stock : 35 bouteille(s)
Commande de 81 bouteille(s)     ->
('127.0.0.1', 60544)
```

```
35 bouteille(s) recue(s)
```

```
Serveur connecté par ('127.0.0.1', 60550)
Livraison de 35 bouteille(s)    ->
('127.0.0.1', 60544)
Reste en stock : 0 bouteille(s)
$ # le serveur s'est arrêté
```

```
Echec commande
[2]-  Fini                    ./wine_bottles_client.py
[3]+  Fini                    ./wine_bottles_client.py
$
```