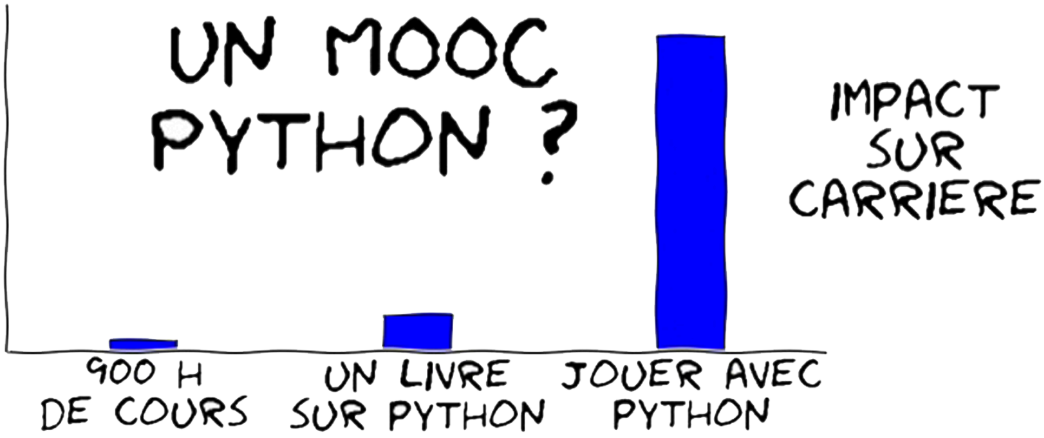
python™ 3

Des fondamentaux au concepts avancés du langage

Session 2 - 17 septembre 2018

Thierry Parmentelat Arnaud Legout



https ://www.fun-mooc.fr

Table des matières

Chapitre 8

Programmation asynchrone avec asyncio

8.1 w8-s1-c1-thread-safety

Pourquoi les threads c'est délicat ?

8.1.1 Complément - niveau avancé

À nouveau dans ce cours nous nous intéressons aux applications qui sont plutôt I/O intensive. Cela dit et pour mettre les choses en perspective, on pourrait se dire que qui peut le plus peut le moins, et que le multi-threading qui est bien adapté au calcul parallèle CPU-intensive, pourrait aussi bien faire l'affaire dans le contexte de l'I/O-intensive.

Il se trouve qu'en fait le multi-theading présente un inconvénient assez notable, que nous allons tenter de mettre en évidence dans ce complément ; sur un exemple hyper-simple, nous allons illustrer la notion de section critique, et montrer pourquoi on doit utiliser parfois - trop souvent - la notion de lock ou verrou lorsqu'on utilise des threads.

8.1.2 avertissement : pas que pour Python

Je dois préciser avant d'aller plus loin que pour cette discussion, nous allons oublier le cas spécifique de Python ; les notions que nous abordons tournent autour des relations entre l'OS et les applications, qui sont valables en général.

En fait c'est même pire que ça, et nous verrons les implications pour Python à la fin du complément ; vous avez peut-être déjà entendu parler du GIL, mais on va avoir besoin d'appréhender cette histoire de section critique pour mieux comprendre les tenants et les aboutissements du GIL en Python.

8.1.3 processus et threads

On rappelle que, pour écrire des programmes parallèles, l'Operating System nous offre principalement deux armes :

— les processus
— les threads

Il faut se souvenir que la première fonction de l'OS est justement que plusieurs programmes puissent s'exécuter en même temps, c'est-à-dire partager les ressources physiques de l'ordinateur, et notamment le CPU et la mémoire, sans pouvoir se contaminer l'un l'autre.

Aussi, c'est par construction que deux processus différents se retrouvent dans des espaces totalement étanches, et qu'un processus ne peut pas accéder à la mémoire d'un autre processus.

On peut naturellement utiliser des processus pour faire du calcul parallèle, mais cette contrainte de naissance rend l'exercice fastidieux, surtout lorsque les différents programmes sont très dépendants les uns des autres, car dans ce cas bien sûr ils ont besoin d'échanger voire de partager des données (je m'empresse de préciser qu'il existe des mécanismes pour faire ça - notamment : librairies de mémoire partagée, envoi de messages - mais qui induisent leur propre complexité…).

Par contraste un processus peut contenir plusieurs threads, chacun disposant pour faire court, d'une pile et d'un pointeur de programme - en gros donc, où on en est dans la logique de une exécution séquentielle ; l'intérêt étant que de tous les threads partagent à présent la mémoire du processus ; c'est donc un modèle a priori très attractif pour notre sujet.

8.1.4 le scheduler

Comme ces notions de processus et de threads sont fournies par l'OS, c'est à lui également que revient la responsabilité de les faire tourner ; cela est fait dans le noyau par ce qu'on appelle le scheduler.

Comment ça marche ? dans le détail, c'est un sujet très copieux, il existe une littérature hyper-abondante sur le sujet, et donc une extrême variété de stratégies et de réglages possibles.

Mais pour ce qui nous intéresse, nous n'allons retenir que ces caractéristiques de haut niveau très simples :

- le scheduler maintient une liste de processus et de threads à faire tourner ;
- il décide - à une fréquence assez élevée - de leur donner la main à tour de rôle ;
- simplement il faut bien réaliser qu'à ce stade, ce que manipule le scheduler, c'est essentiellement du code binaire, très proche du processeur, après toutes les phases de compilation et optimisation.

context switches

L'instant où le scheduler décide de suspendre l'exécution d'un processus - ou thread - pour donner la main à un autre, s'appelle un context switch ; on parle de process switch* lorsqu'on passe d'un processus à un autre, et de task switch ou thread switch lorsqu'on passe d'un thread à un autre à l'intérieur d'un processus.

Le point important pour nous, c'est que le scheduler est un morceau de code générique, il fait donc son travail de manière neutre pour tous les processus ou threads, indépendamment du langage par exemple, ou du domaine d'application ; et que le découpage du temps en slots alloués aux différents joueurs se fait bien évidemment sur la base des instuctions élémentaires du processeur - ce qu'on appelle les cycles.

On s'intéresse davantage aux threads dans la suite, et nous allons voir que dans ce cas, cela crée parfois de mauvaises surprises.

8.1.5 une simple opération d'addition

Pour illustrer notre propos, nous allons étudier une opération extrêmement banale qui consiste à incrémenter la valeur d'une variable.

Il se trouve qu'en pratique cette opération se décompose en réalité en 3 opérations élémentaires, comme le montre la figure suivante ; à nouveau le langage utilisé dans toutes ces illustrations n'est pas du Python - typiquement une opération comme celle-ci en Python va occasionner bien plus d'instructions élémentaires que cela - disons pour fixer les idées que c'est quelque chose comme du C ; peu importe en fait, c'est l'idée qui est importante.

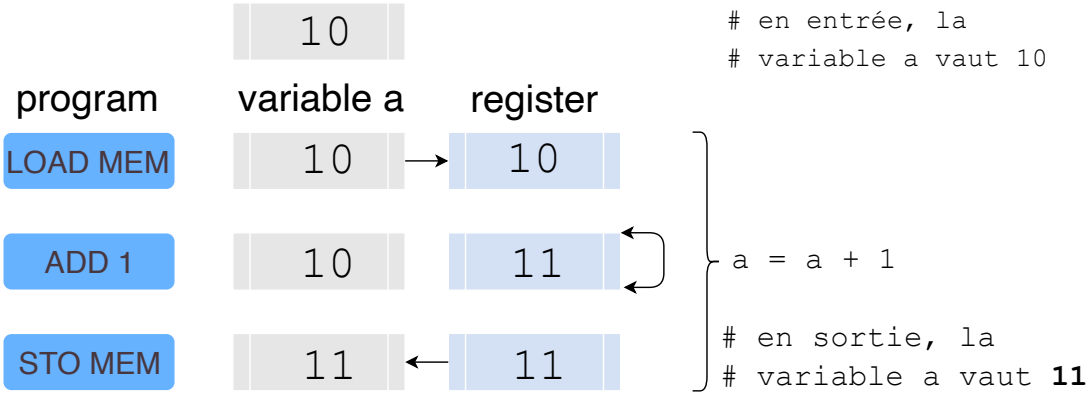


Figure 1 : une instruction du genre de `a = a + 1` dans un langage compilé, avec un seul thread

---

On voit sur cette figure la logique des trois opérations * dans un premier temps on va chercher la valeur de la variable `a` qu'on range disons dans un registre - ou un cache ; * on réalise l'incrémentation de cette valeur dans le registre * puis on recopie le résultat dans la case mémoire originelle, qui correspond à la variable `a`

Ce programme fait donc bien ce qu'on veut ; si la valeur de `a` était 10 au début, on y trouve `11` à la fin, tout va bien.

8.1.6 dans deux threads, un scénario favorable

À présent, nous allons imaginer le cas de deux threads qui s'exécutent en parallèle, avec un seul processeur ;
et admettons que chacun des deux threads exécute une fois `a = a + 1` sur une variable globale `a`.

En admettant comme tout à l'heure que `a` valait `10` en commençant, on s'attend donc naturellement à ce qu'à la fin `a` vaille `12` puisqu'on l'aura incrémenté deux fois.

Voyons d'abord un scénario qui se passe bien ; le scheduler qui, donc, donne la main alternativement à l'un et l'autre de nos deux threads, a la bonne idée de laisser intègres les deux blocs de 3 instructions, sans y insérer de context switching.

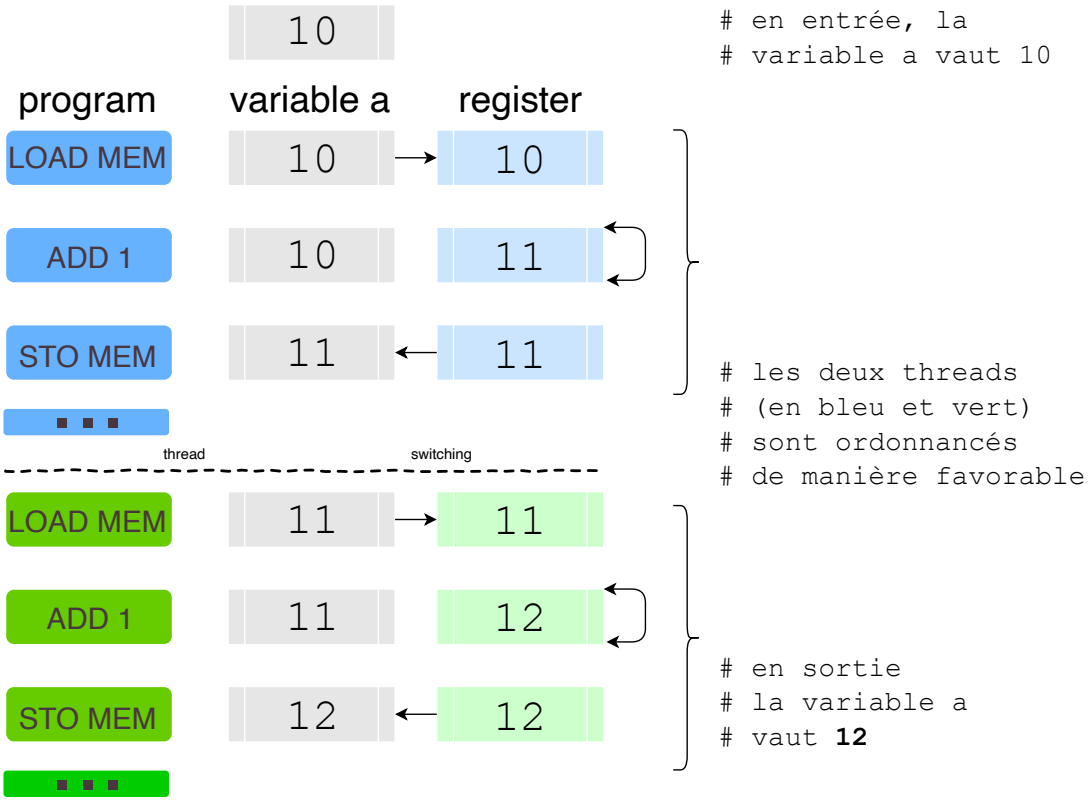


Figure 2 : deux threads, un processeur, dans un scénario favorable

---

Dans ce scénario, à l'issue des deux threads on a bien, comme attendu, `a == 12`.

8.1.7 toujours 2 threads, mais pas de chance

Mais en fait, il y a un souci avec cette façon de faire.

Ce qu'il faut bien comprendre c'est que du point de vue du scheduler, toutes ces instructions sont pareilles et il n'y a pas, à ce stade, de moyen pour le scheduler, de traiter ce bloc de 3 instructions de manière particulière.

Aussi le scheduler, qui a déjà un travail assez compliqué si on tient compte du fait qu'il doit être fair (donner autant de temps à tout le monde), choisit les points de context switching comme il le peut au milieu de ce qui, pour lui, n'est qu'une longue liste d'instructions.

Imaginons du coup un scénario moins favorable que le précédent, dans lequel le scheduler, pas de chance, choisit de faire un context switching juste après le premier LOAD du premier thread ; ça nous donne alors l'exécution décrite dans cette figure :

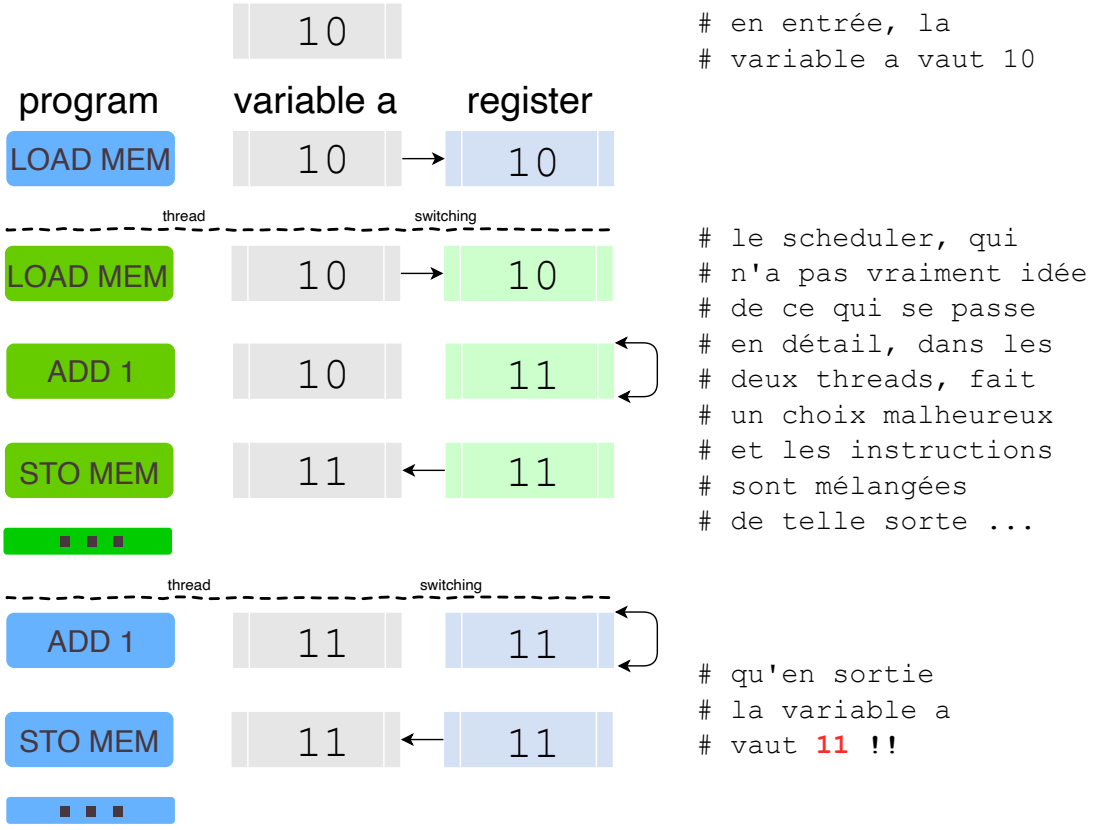


Figure 3 : deux threads, un processeur, mais un choix de scheduling malheureux

---

Du coup ce qui se passe ici, c'est que le deuxième fil fait son LOAD à partir de la variable `a` qui n'a pas encore été modifiée, et du coup les deux threads incrémentent tous les deux la valeur 10, et à l'issue de l'exécution des deux threads, on a maintenant `a == 11` !!

Pour résumer donc : on part de `a == 10`, on exécute 2 threads qui font tous les deux `a = a + 1` et au final, on se retrouve avec `a == 11` ; gros souci donc !

8.1.8 phénomène général

À ce stade vous pourriez vous dire que j'ai triché, et que j'ai choisi un scénario irréaliste ; par exemple qu'en pratique l'incrémentation de 1 ça se fait en hardware en une seule instruction.

Oui bien sûr, l'exemple est choisi pour rester aussi simple que possible ; mais si vous n'êtes pas convaincu avec `a = a + 1`, prenez simplement `a = a + b`, vous verrez que c'est exactement le même souci.

En fait le souci que l'on a, de manière générale, c'est que :

- dans un langage de programmation un tout petit peu évolué, un fragment de code (même réduit à une instruction) se traduit presque toujours en plusieurs instructions binaires pour le processeur
- pour que le programme fonctionne correctement dans un mode multi-thread, certains fragments de code, et notamment ceux qui accèdent à de la mémoire partagée, doivent être exécutés de façon atomique (c-à-d ne pas être interrompus en plein milieu par le scheduler)
- et sans aide du programmeur, le scheduler n'a aucun moyen de savoir où, dans le flot d'instruction binaires, il est légitime ou pas de faire un context switching.

Et avec quelque chose d'un tout petit peu plus compliqué comme `a = 2 * a; a = a + 1`, on n'a même pas besoin de descendre au niveau du code machine pour exhiber le problème…

8.1.9 verrou et exclusion mutuelle

Du coup, pour rendre la programmation par thread utilisable en pratique, il faut lui adjoindre des mécanismes, accessibles au programmeur, pour rendre explicite ce type de problèmes.

La notion la plus simple de ces mécanismes est celle de verrou pour implémenter une exclusion mutuelle ; pour en donner une illustration très rapide, voyons cela sur notre exemple.

Nous allons remplacer ceci :

```
# thread A
a = a + 1

# thread B
a = a + 1
```

par ceci

```
# thread A

get_lock(lock)
a = a + 1
release_lock(lock)

# thread B

get_lock(lock)
a = a + 1
release_lock(lock)
```

Dans cette nouvelle version, un nouvel objet global `lock` est introduit, qui peut être dans deux états libre ou occupé.

De cette façon, celui des deux threads qui arrive à ce stade en premier obtient le verrou (le met dans l'état occupé), et fait son traitement avant de le relâcher ; du coup l'autre doit attendre que le premier ait fini tout le traitement de sa section critique pour pouvoir commencer le sien.

Comme on le voit, l'idée consiste à permettre au programmeur de rendre explicite l'exclusion mutuelle qu'il est nécessaire d'assurer pour que le programme fonctionne comme prévu, et de façon déterministe.

8.1.10 ce qu'il faut retenir

Pour conclure cette partie, retenons que l'on peut écrire du code multi-thread dont le comportement est déterministe, mais au prix de l'ajout dans le code d'annotations qui limitent les modes d'exécution ; ce qui a tendance à rendre les choses complexes, et donc coûteuses.

Et retenons que le problème principal ici est lié à l'absence de contrôle, par le programmeur, sur les context switchings ; et du coup ceux-ci peuvent intervenir à n'importe quel moment.

Nous verrons que la situation est très différente avec le paradigme `async/await/asyncio`.

8.1.11 le cas de Python : le GIL

Dans ce contexte, le cas des programmes Python est un peu spécial ; ce n'est pas un langage compilé, ce qui signifie que du point de vue de l'OS et du scheduler, le processus qui tourne est en fait l'interpréteur Python.

Et il se trouve que l'interpréteur Python est un exemple de programme qui pourrait être sensible au type de problèmes que nous venons d'étudier.

Voyons un exemple pour vous faire entrevoir la complexité du sujet. Vous vous souvenez qu'on a parlé de garbage collection, et de compteur de références. Voyons comment le fait de maintenir un compteur de références crée le besoin d'écrire dans la mémoire, alors qu'en lisant le code Python on ne voit que des accès en lecture.

[1]:
```
# on est bien d'accord que ce code ne fait que lire
# le contenu de x et ne modifie pas sa valeur

def foo(x,  max_depth, depth=1):
    print(f"in {depth}th function call we see {x}")
    if depth < max_depth:
        foo(x, max_depth, depth+1)
```

[2]:
```
# j'exécute ce code sur un objet tout neuf
a = []

# et je confirme bien qu'on n'y touche jamais
foo(a, 3)
```

```
in 1th function call we see []
in 2th function call we see []
in 3th function call we see []
```

[3]:
```
# mais en fait pendant toute l'exécution de ce code
# il y a des changements qui sont faits dans l'objet a
# en tous cas dans sa représentation interne,
# regardons notamment le compteur de références

# pour importer getrefcount() qui permet de
# lire le compteur de références d'un objet
import sys

# la même logique exactement que plus haut,
# mais ici affiche aussi le compteur de références
def bar(x,  max_depth, depth=1):
    print(f"in {depth}th function call we see {x} that has {sys.getrefcount(x)}␣
 ↪refs")
    if depth < max_depth:
        bar(x, max_depth, depth+1)

bar(a, 3)
```

```
in 1th function call we see [] that has 4 refs
in 2th function call we see [] that has 6 refs
in 3th function call we see [] that has 8 refs
```

Du coup, et précisément pour protéger son fonctionnement intime, l'interpréteur Python est implémenté de telle sorte à empêcher l'exécution simultanée de plusieurs threads dans un même processus Python ! Cela est fait au travers d'un verrou central pour tout l'interpréteur, qui s'appelle le GIL - le Global Interpreter Lock.

Aussi, bien qu'il est possible - notamment au travers de la librairie `threading` - de concevoir des programmes multi-threadés en Python, par construction, ils ne peuvent pas s'exécuter en parallèle, et notamment ne peuvent pas tirer profit d'une architecture multi-processeur (pour cela en Python, il ne reste que l'option multi-processus). Ce qui, il faut bien l'admettre, ruine un peu l'intérêt…

8.1.12 pour en savoir plus

Cette présentation est juste une mise en perspective, elle est volontairement superficielle, d'autant que nous sommes clairement à côté de notre sujet. Si vous souhaitez approfondir certains de ces points (malheureusement sur ces sujets pointus les sources anglaises, même sur wikipedia, sont souvent préférables…) :

- sur la notion de thread : Thread(computing) on wikipedia
- sur le mécanisme de verrou :
  - une illustration en vidéo (1'17) du verrou, très proche de notre exemple
  - wikipedia en anglais
- sur le GIL : `https://realpython.com/python-gil/`

---

remarque à propos des verrous

Le lecteur attentif remarquera une contradiction apparente, car dans notre présentation des verrous, on a introduit … un nouvel objet global `lock` ; on pourrait craindre de n'avoir fait ici que de reporter le problème. N'aurait-on pas seulement déplacé le souci qu'on avait avec globale `a` sur la globale `lock` ?

En réalité ça n'est pas le cas, cette approche fonctionne vraiment et est massivement utilisée. Elle fonctionne notamment parce que le mécanisme de verrou est cette fois connu du scheduler, qui par conséquent peut garantir le comportement de `get_lock()` et `release_lock()`.

8.2 w8-s1-c2-warning-python37

Avertissement relatif à **asyncio** et Python-3.7

8.2.1 Complément - niveau intermédiaire

Puisque cette semaine est consacrée à **asyncio**, il faut savoir que cette brique technologique est relativement récente, et qu'elle est du coup, plus que d'autres aspects de Python, sujette à des évolutions.

8.2.2 Les vidéos utilisent Python-3.6

Comme on l'a dit en préambule du cours, la version de référence lors du tournage était Python-3.6. Par contre les notebooks sur FUN-MOOC utilisent à présent une version plus récente.

[1]:
```
import sys
major, minor, *_ = sys.version_info
print(f"les notebooks utilisent la version {major}.{minor}")
```

```
les notebooks utilisent la version 3.10
```

8.2.3 Un résumé des nouveautés

Vous trouverez à la fin de la semaine, dans la séquence consacrée aux bonnes pratiques, un résumé des améliorations apportées depuis la version 3.6.

8.2.4 L'essentiel est toujours d'actualité

Cela étant dit, nos buts ici étaient principalement :

- de vous faire découvrir ce nouveau paradigme,
- de vous faire sentir dans quelles applications cela peut avoir un apport très précieux,
- de bien vous faire comprendre ce qui se passe à l'exécution,
- et de vous donner un aperçu de la façon dont tout cela est implémenté.

8.2.5 Les différences les plus visibles

Les plus grosses différences concernent la prise en main. Comme nous allons bientôt le voir, le "hello world" de `asyncio` était en Python-3.6 un peu awkward, cela nécessitait pas mal de circonlocutions.

C'est-à-dire que pour faire fonctionner la coroutine :

[2]:
```
# un exemple de coroutine
import asyncio

async def hello_world():
    await asyncio.sleep(0.2)
    print("Hello World")
```

En Python-3.6

Pour exécuter cette coroutine dans un interpréteur Python-3.6, la syntaxe est un peu lourdingue :

```
# pour exécuter uniquement cette coroutine en Python-3.6
loop = asyncio.get_event_loop()
loop.run_until_complete(hello_world())
```

En Python-3.7

En 3.7, on arrive au même résultat de manière beaucoup plus simple :

```
# c'est beaucoup plus simple en 3.7
asyncio.run(hello_world())
```

Avec IPython 7

Notez qu'avec IPython (et donc aussi dans les notebooks) c'est encore plus simple ; en effet IPython s'est débrouillé pour autoriser la syntaxe suivante :

[3]:
```
# depuis ipython, ou dans un notebook, vous pouvez faire simplement

await hello_world()
```

```
Hello World
```

Mise en garde attention toutefois, je vous mets en garde contre le fait que ceci est une commodité pour nous faciliter la vie, mais elle est spécifique à IPython et ne va pas fonctionner tel quel dans un programme exécuté directement par l'interpréteur Python standard.

[4]:
```
# un code cassé

!cat data/broken-await.py
```

```
import asyncio

async def hello_world():
    await asyncio.sleep(0.2)
    print("Hello World")

# ceci ne fonctionne pas
```

```
# en Python standard
await hello_world()
```

[5]:
```
# la preuve

!python data/broken-await.py
```

```
  File "/Users/tparment/git/flotpython-tools/pdf/work/data/broken-await.py"
   , line 9
    await hello_world()
    ^^^^^^^^^^^^^^^^^^^
SyntaxError: 'await' outside function
```

Nous avons choisi de ne pas utiliser ce trait dans les notebooks, car cela pourrait créer de la confusion, mais n'hésitez pas à l'utiliser de votre côté une fois que tout ceci est bien acquis.

À propos de Python-3.8

Avec Python 3.8 et 3.9 il y a peu de changements concernant `asyncio`, ils sont décrits ici :

https://docs.python.org/3/whatsnew/3.8.html#asyncio https://docs.python.org/3/whatsnew/3.9.html#asyncio

Notez toutefois l'apparition en 3.8 d'une REPL (read-eval-print-loop) qui supporte justement `await` au toplevel

8.2.6 Conclusion

Pour conclure cet avertissement, ne vous formalisez pas si vous voyez dans le cours des pratiques qui sont dépassées. Les différences par rapport aux pratiques actuelles - même si elles sont assez visibles dans ce cours introductif - sont en réalité mineures au niveau de ce qu'il est important de comprendre quand on aborde d'un oeil neuf ce nouveau paradigme de programmation.

8.3 w8-s4-c1-async-http

Essayez vous-même

8.3.1 Complément - niveau avancé

Pour des raisons techniques, il ne nous est pas possible de mettre en ligne un notebook qui vous permette de reproduire les exemples de la vidéo.

C'est pourquoi, si vous êtes intéressés à reproduire vous-même les expériences de la vidéo - à savoir, aller chercher plusieurs URLs de manière séquentielle ou en parallèle - vous pouvez télécharger le code fourni dans ce lien.

Il s'agit d'un simple script, qui reprend les 3 approches de la vidéo :

- accès en séquence ;
- accès asynchrones avec `fetch` ;
- accès asynchrones avec `fetch2` (qui pour rappel provoque un tick à chaque ligne qui revient d'un des serveurs web).

À part pour l'appel à `sys.stdout.flush()`, ce code est rigoureusement identique à celui utilisé dans la vidéo. On doit faire ici cet appel à `flush()`, dans le mode avec `fetch2`, car sinon les sorties de notre script sont bufferisées, et apparaissent toutes ensemble à la fin du programme, c'est beaucoup moins drôle.

Voici son mode d'emploi :

```
$ python3 async_http.py --help
usage: async_http.py [-h] [-s] [-d] [urls [urls ...]]

positional arguments:
  urls              URL's to be fetched

optional arguments:
  -h, --help        show this help message and exit
  -s, --sequential  run sequentially
  -d, --details     show details of lines as they show up (using fetch2)
```

Et voici les chiffres que j'obtiens lorsque je l'utilise dans une configuration réseau plus stable que dans la vidéo, on voit ici un réel gain à l'utilisation de communications asynchrones (à cause de conditions réseau un peu erratiques lors de la vidéo, on n'y voit pas bien le gain obtenu) :

```
$ python3 async_http.py -s
Running sequential mode on 4 URLs
http://www.irs.gov/pub/irs-pdf/f1040.pdf returned 179940 chars
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf returned 113242 chars
http://www.irs.gov/pub/irs-pdf/f1040es.pdf returned 395201 chars
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf returned 73189 chars
duration = 9.80829906463623s
```

```
$ python3 async_http.py
Running simple mode (fetch) on 4 URLs
fetching http://www.irs.gov/pub/irs-pdf/f1040.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040sb.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040es.pdf
fetching http://www.irs.gov/pub/irs-pdf/f1040ez.pdf
http://www.irs.gov/pub/irs-pdf/f1040.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040es.pdf response status 200
http://www.irs.gov/pub/irs-pdf/f1040sb.pdf returned 75864 bytes
http://www.irs.gov/pub/irs-pdf/f1040.pdf returned 186928 bytes
http://www.irs.gov/pub/irs-pdf/f1040ez.pdf returned 117807 bytes
http://www.irs.gov/pub/irs-pdf/f1040es.pdf returned 409193 bytes
duration = 2.211031913757324s
```

N'hésitez pas à utiliser ceci comme base pour expérimenter.

Nous verrons en fin de semaine un autre exemple qui cette fois illustrera l'interaction avec les sous-processus.

8.4 w8-s8-c1-players

asyncio - un exemple un peu plus réaliste

8.4.1 Complément - niveau avancé

Pour des raisons techniques, il n'est pas possible de mettre en ligne un notebook pour les activités liées au réseau, qui sont pourtant clairement dans le coeur de cible de la bibliothèque - souvenez-vous que ce paradigme de programmation a été développé au départ par les projets comme tornado, qui se préoccupe de services Web.

Aussi, pour illustrer les possibilités offertes par `asyncio` sur un exemple un peu plus significatif que ceux qui utilisent `asyncio.sleep`, nous allons écrire le début d'une petite architecture de jeu.

Il s'agit pour nous principalement d'illustrer les capacités de `asyncio` en matière de gestion de sous-processus, car c'est quelque chose que l'on peut déployer dans le contexte des notebooks.

Nous allons procéder en deux temps. Dans ce premier notebook nous allons écrire un petit programme Python qui s'appelle `players.py`. C'est une brique de base dans notre architecture, dans le second notebook on écrira un programme qui lance (sous la forme de sous-processus) plusieurs instances de `players.py`.

Le programme `players.py`

Mais dans l'immédiat, voyons ce que fait `players.py`. On veut modéliser le comportement de plusieurs joueurs.

Chaque joueur a un comportement hyper basique, il émet simplement à des intervalles aléatoires un événement du type :

> je suis le joueur John et je vais dans la direction Nord

Chaque joueur a un nom, et une fréquence moyenne, et un nombre de cycles.

Par ailleurs pour être un peu vraisemblable, il y a quatre directions `N`, `S`, `E` et `W`, mais que l'on n'utilisera pas vraiment dans la suite.

Voyez ici le code de `players.py`

Comme vous le voyez, dans ce premier exemple nous n'utilisons à nouveau que `asyncio.sleep` pour modéliser chaque joueur, dont la logique peut être illustrée simplement comme ceci (où la ligne horizontale représente le temps) :

configurations prédéfinies

Pour éviter de nous noyer dans des configurations compliquées, on a embarqué dans `players` plusieurs (4) configurations prédéfinies - voyez la globale `predefined`.

Dans tous les cas chacune de ces configurations crée deux joueurs.

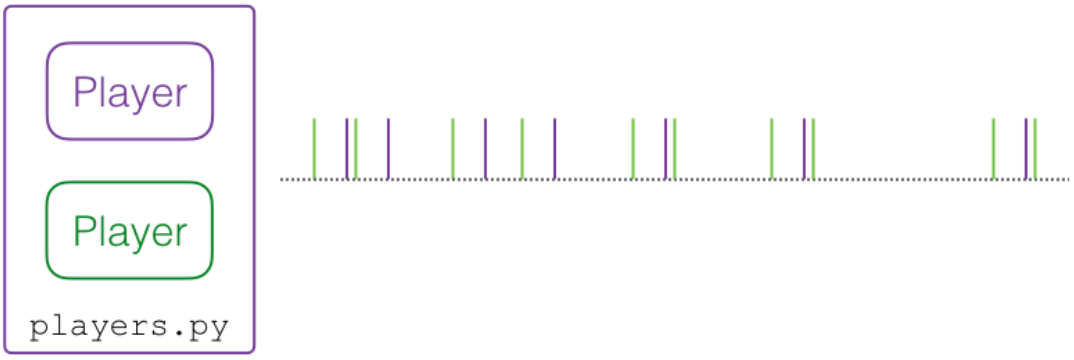
[1]:
```
# par exemple la config. prédéfinie # 1
# ressemble à ceci
from data.players import predefined

for predef, players in predefined.items():
    print(f"predefined {predef}: {players}")
```

```
predefined 1: [Players john: 3x0.8s + mary: 7x0.4s]
predefined 2: [Players bill: 5x0.5s + jane: 4x0.7s]
predefined 3: [Players augustin: 8x0.8s + randalph: 10x0.6s]
predefined 4: [Players bertrand: 12x0.5s + juliette: 8x0.7s]
```

Ce qui signifie qu'avec la config. #1, on génère 3 événements pour `john`, et 7 pour `mary`; et la durée entre les événements de `john` est tirée au hasard entre 0 et 0.8s.

La logique des deux joueurs est simplement juxtaposée, ou si on préfère superposée, par `asyncio.gather`, ce qui fait que la sortie de `players.py` ressemble à ceci :



[2]:
```
# je peux lancer un sous-processus
# depuis le notebook
# ici la config #1
!data/players.py
```

```
/Users/tparment/git/flotpython-tools/pdf/work/data/players.py:77: Deprecati
   onWarning: There is no current event loop
  loop = asyncio.get_event_loop()
N john
E mary
E mary
W mary
W mary
W mary
S john
S mary
N mary
N john
```

[3]:
```
# ou une autre configuration
!data/players.py 2
```

```
/Users/tparment/git/flotpython-tools/pdf/work/data/players.py:77: Deprecati
   onWarning: There is no current event loop
  loop = asyncio.get_event_loop()
S bill
N jane
N bill
N bill
N jane
E jane
W bill
N bill
W jane
```

Nous allons voir dans le notebook suivant comment on peut orchestrer plusieurs instances du programme `players.py`, et prolonger cette logique de juxtaposition / mélange des sorties, mais cette fois au niveau de plusieurs processus.

8.5 w8-s8-c2-game

Gestion de sous-process

8.5.1 Complément - niveau avancé

Dans ce second notebook, nous allons étudier un deuxième programme Python, que j'appelle `game.py` (en fait c'est le présent notebook).

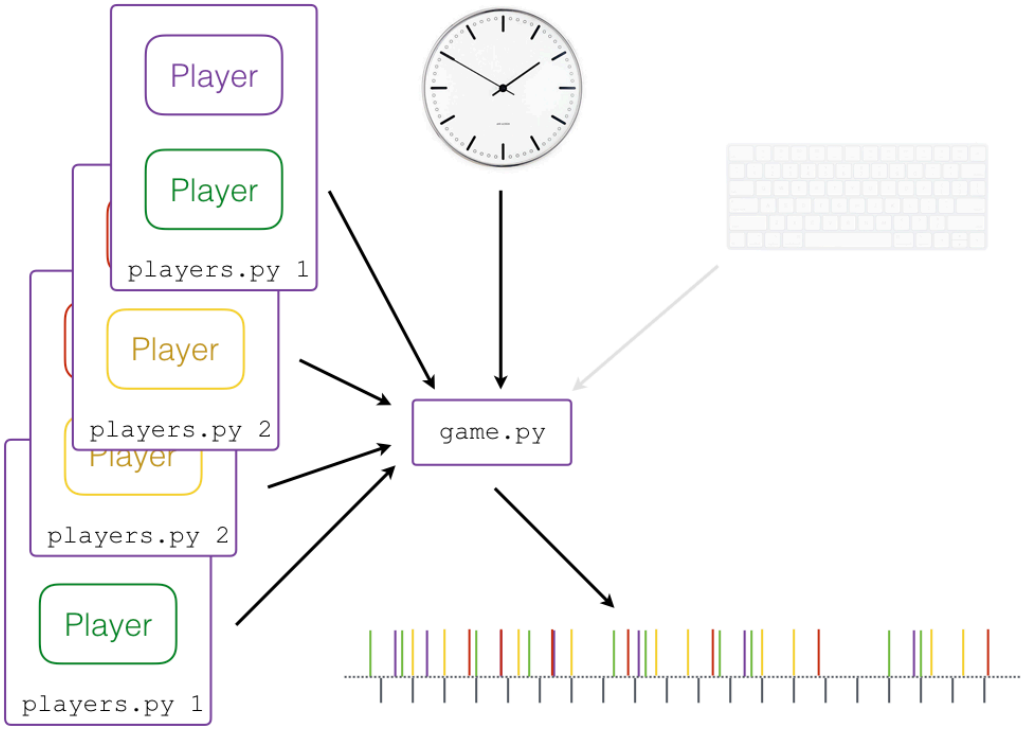
Fonctions de `game.py`

Son travail va consister à faire plusieurs choses en même temps ; pour rester le plus simple possible, on va se contenter des trois fonctions suivantes :

- scheduler (chef d'orchestre) : on veut lancer à des moments préprogrammés des instances (sous-process) de `players.py` ;
- multiplexer (agrégateur) : on veut lire et imprimer au fur et à mesure les messages émis par les sous-processus ;
- horloge : on veut également afficher, chaque seconde, le temps écoulé depuis le début.

En pratique, le programme `game.py` serait plutôt le serveur du jeu qui reçoit les mouvements de tous les joueurs, et diffuse ensuite en retour, en mode broadcast, un état du jeu à tous les participants.

Mais dans notre version hyper simpliste, ça donne un comportement que j'ai essayé d'illustrer comme ceci :



Remarque concernant les notebooks et le clavier

Lorsqu'on exécute du code Python dans un notebook, les entrées clavier sont en fait interceptées par le navigateur Web ; du coup on ne peut pas facilement (du tout ?) faire tourner dans un notebook un programme asynchrone qui réagirait aussi aux événements de type entrée clavier.

C'est pour cette raison que le clavier apparaît sur ma figure en filigrane. Si vous allez jusqu'à exécuter ce notebook localement sur votre machine (voir plus bas), vous pourrez utiliser le clavier pour ajouter à la volée des éléments dans le scénario ; il vous suffira de taper au clavier un numéro de 1 à 4 (suivi de Entrée) au moment où voulez ajouter une paire de joueurs, dans une des 4 configurations prédéfinies de `players.py`.

Terminaison

On choisit de terminer le programme `game.py` lorsque le dernier sous-processus `players.py` se termine.

Le programme `game.py`

C'est ce notebook qui va jouer pour nous le rôle du programme `game.py`.

[1]:
```
import asyncio
import sys
```

[2]:
```
# cette constante est utile pour déclarer qu'on a l'intention
# de lire les sorties (stout et stderr)
# de nos sous-process par l'intermédiaire de pipes
from subprocess import PIPE
```

Commençons par la classe `Scheduler` ; c'est celle qui va se charger de lancer les sous-process selon un scénario configurable. Pour ne pas trop se compliquer la vie on choisit de représenter un scénario (un script) comme une liste de tuples de la forme :

```
script = [ (time, predef), ...]
```

qui signifie de lancer, un délai de `time` secondes après le début du programme, le programme `players.py` dans la configuration `predef` - toujours de 1 à 4 donc.

[3]:
```
class Scheduler:

    def __init__(self, script):

        # on trie le script par ordre chronologique
        self.script = list(script)
        self.script.sort(key = lambda time_predef : time_predef[0])

        # juste pour donner un numéro à chaque processus
        self.counter = 1
        # combien de processus sont actifs
        self.running = 0
        # nombre de processus à executer et nombre de processus terminés
        self.to_be_run = len(script)
        self.finished = 0


    async def run(self):
        """
        fait tout le travail, c'est-à-dire :
        * lance tous les sous-processus à l'heure indiquée
        * et aussi en préambule, pour le mode avec clavier,
          arme une callback sur l'entrée standard
        """
        # pour le mode avec clavier (pas fonctionnel dans le notebook)
```

```
        # on arme une callback sur stdin
        asyncio.get_event_loop().add_reader(
            # il nous faut un file descriptor, pas un objet Python
            sys.stdin.fileno(),
            # la callback
            Scheduler.read_keyboard_line,
            # les arguments de la callback
            # cette fois c'est un objet Python
            self, sys.stdin
        )

        # le scénario prédéfini
        epoch = 0
        for tick, predef in self.script:
            # attendre le bon moment
            await asyncio.sleep(tick - epoch)
            # pour le prochain
            epoch = tick
            asyncio.ensure_future(self.fork_players(predef))


    async def fork_players(self, predef):
        """
        lance maintenant une instance de players.py avec cette config

        puis
        écoute à la fois sdtout et stderr, et les imprime
        (bon c'est vrai que players n'écrit rien sur stderr)
        attend la fin du sous-processus (avec wait())
        et retourne son code de retour (exitcode) du sous-processus

        par commodité on décide d'arrêter la boucle principale
        lorsqu'il n'y a plus aucun process actif
        """

        # la commande à lancer pour forker une instance de players.py
        # l'option python -u sert à désactiver le buffering sur stdout
        command = f"python3 -u data/players.py {predef}".split()

        # pour afficher un nom un peu plus parlant
        worker = f"ps#{self.counter} (predef {predef})"

        # housekeeping
        self.counter += 1
        self.running += 1

        # c'est là que ça se passe : on forke
        print(8 * '>', f"worker {worker}")
        process = await asyncio.create_subprocess_exec(
            *command,
            stdout=PIPE, stderr=PIPE,
        )
        # et on lit et écrit les canaux du sous-process
        stdout, stderr = await asyncio.gather(
            self.read_and_display(process.stdout, worker),
            self.read_and_display(process.stderr, worker))
        # qu'il ne faut pas oublier d'attendre pour que l'OS sache
```

```
        # qu'il peut nettoyer
        retcod = await process.wait()

        # le process est terminé
        self.running -= 1
        self.finished += 1
        print(8 * '<', f"worker {worker} - exit code {retcod}"
              f" - {self.running} still running")

        # si c'était le dernier on sort de la boucle principale
        # if self.running == 0:
        # Plus de processus en cours d'execution ET tous les processus executés
        if self.running == 0 and self.finished == self.to_be_run:
            print("no process left - bye")
            asyncio.get_event_loop().stop()
        # sinon on retourne le code de retour
        return retcod


    async def read_and_display(self, stream, worker):
        """
        une coroutine pour afficher les sorties d'un canal
        stdout ou stderr d'un sous-process
        elle retourne lorsque le processus est terminé
        """
        while True:
            bytes = await stream.readline()
            # l'OS nous signale qu'on en a terminé
            # avec ce process en renvoyant ici un objet bytes vide
            if not bytes:
                break

            # bien qu'ici players n'écrit que de l'ASCII
            # readline() nous renvoie un objet `bytes`
            # qu'il faut convertir en str
            line = bytes.decode().strip()
            print(8 * ' ', f"got `{line}` from {worker}")


    # ceci est seulement fonctionnel si vous exécutez
    # le programme localement sur votre ordinateur
    # car depuis un notebook le clavier est intercepté
    # par le serveur web
    def read_keyboard_line(self, stdin):
        """
        ceci est une callback; eh oui :)
        c'est pourquoi d'ailleurs ce n'est pas une coroutine
        cependant on est sûr qu'elle n'est appelée
        que lorsqu'il y a réellement quelque chose à lire
        """
        line = stdin.readline().strip()
        # ici je triche complètement
        # lorsqu'on est dans un notebook, pour bien faire
        # on ne devrait pas regarder stdin du tout
        # mais pour garder le code le plus simple possible
        # je choisis d'ignorer les lignes vides ici
        # comme ça mon code marche dans les deux cas
```

```
        if not line:
            return
        # on traduit la ligne tapée au clavier
        # en un entier entre 1 et 4
        try:
            predef = int(line)
            if not (1 <= predef <= 4):
                raise ValueError('entre 1 et 4')
        except Exception as e:
            print(f"{line} doit être entre 1 et 4 {type(e)} - {e}")
            return
        asyncio.ensure_future(self.fork_players(predef))
        # Un nouveau processus à executer
        self.to_be_run += 1
```

À ce stade on a déjà le cœur de la logique du scheduler, et aussi du multiplexer. Il ne nous manque plus que l'horloge :

[4]:
```
class Clock:

    def __init__(self):
        self.clock_seconds = 0

    async def run(self):
        while True:
            print(f"clock = {self.clock_seconds:04d}s")
            await asyncio.sleep(1)
            self.clock_seconds += 1
```

Et enfin pour mettre tous ces morceaux en route il nous faut une boucle d'événements :

[5]:
```
class Game:

    def __init__(self, script):
        self.script = script

    def mainloop(self):
        loop = asyncio.get_event_loop()

        # on met ensemble une clock et un scheduler
        clock = Clock()
        scheduler = Scheduler(self.script)

        # et on fait tourner le tout
        asyncio.ensure_future(clock.run())
        asyncio.ensure_future(scheduler.run())
        loop.run_forever()
```

Et maintenant je peux lancer une session simple ; pour ne pas être noyé par les sorties on va se contenter de lancer :

— 0.5 seconde après le début une instance de `players.py 1`
— 1 seconde après le début une instance de `players.py 2`

[6]:
```
# nous allons juxtaposer 3 instances de players.py
# et donc avoir 6 joueurs dans le jeu
```

```
# La dernière instance se déroulera alors que les 2 premières sont terminées
game = Game( [(0.5, 1), (1., 2), (6., 3)])
```

[7]:
```
# si vous êtes dans un notebook
# cette exécution fonctionne, mais pour de sombres raisons
# liées à des évolutions de IPython, le kernel va mourir
# à la fin; ce n'est pas important..
game.mainloop()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[7], line 5
      1 # si vous êtes dans un notebook
      2 # cette exécution fonctionne, mais pour de sombres raisons
      3 # liées à des évolutions de IPython, le kernel va mourir
      4 # à la fin; ce n'est pas important..
----> 5 game.mainloop()

Cell In[5], line 16, in Game.mainloop(self)
     14 asyncio.ensure_future(clock.run())
     15 asyncio.ensure_future(scheduler.run())
---> 16 loop.run_forever()

File ~/miniconda3/envs/flotpython-course/lib/python3.10/asyncio/base_events.py:592,␣
 ↪in BaseEventLoop.run_forever(self)
    590 """Run until stop() is called."""
    591 self._check_closed()
--> 592 self._check_running()
    593 self._set_coroutine_origin_tracking(self._debug)
    595 old_agen_hooks = sys.get_asyncgen_hooks()

File ~/miniconda3/envs/flotpython-course/lib/python3.10/asyncio/base_events.py:584,␣
 ↪in BaseEventLoop._check_running(self)
    582 def _check_running(self):
    583     if self.is_running():
--> 584         raise RuntimeError('This event loop is already running')
    585     if events._get_running_loop() is not None:
    586         raise RuntimeError(
    587             'Cannot run the event loop while another loop is running')

RuntimeError: This event loop is already running
```

Conclusion

Notre but avec cet exemple est de vous montrer, après les exemples des vidéos qui reposent en grande majorité sur `asyncio.sleep`, que la boucle d'événements de `asyncio` permet d'avoir accès, de manière simple et efficace, à des événements de niveau OS. Dans un complément précédent nous avions aperçu la gestion de requêtes HTTP ; ici nous avons illustré la gestion de sous-process.

Actuellement on peut trouver des bibliothèques au dessus de `asyncio` pour manipuler de cette façon quasiment tous les protocoles réseau, et autres accès à des bases de données.

Exécution en local

Si vous voulez exécuter ce code localement sur votre machine :

Tout d'abord sachez que je n'ai pas du tout essayé ceci sur un OS Windows - et d'ailleurs ça m'intéresserait assez de savoir si ça fonctionne ou pas.

Cela étant dit, il vous suffit alors de télécharger le présent notebook au format Python. Vous aurez aussi besoin :

— du code de `players.py`, évidemment ;
— et de modifier le fichier téléchargé pour lancer `players.py` au lieu de `data/players.py`, qui ne fait de sens probablement que sur le serveur de notebooks.

Comme on l'a indiqué plus haut, si vous l'exécutez en local vous pourrez cette fois interagir aussi via la clavier, et ajouter à la volée des sous-process qui n'étaient pas prévus initialement dans le scénario.

8.5.2 Pour aller plus loin

Je vous signale enfin, si vous êtes intéressés à creuser encore davantage, ce tutorial intéressant qui implémente un jeu complet.

Naturellement ce tutorial est lui basé sur du code réseau et non, comme nous y sommes contraints, sur une architecture de type sous-process ; le jeu en question est même en ligne ici...

8.6 w8-s9-c1-news-python37

Nouveautés par rapport aux vidéos

8.6.1 Complément - niveau intermédiaire

Comme on l'a signalé au début de la semaine, `asyncio` a subi quelques modifications dans Python-3.7, que nous allons rapidement illustrer dans ce complément.

Nous verrons aussi par ailleurs une curiosité liée à la dernière version de IPython, qui vise à faciliter le debug et la mise au point de code asynchrone.

Python-3.7 et `asyncio`

Documentation L'évolution la plus radicale est une refonte totale de la documentation.

C'est une très bonne nouvelle, car de l'aveu même de Guido van Rossum, la documentation en place pour les versions 3.5 et 3.6 était particulièrement obscure ; voici comment il l'a annoncé :

> Finally the asyncio docs are not an embarrassment to us all.

Si vous avez déjà eu l'occasion de parcourir ces anciennes documentations, et que vous les avez trouvées indigestes, sachez que vous n'êtes pas seul dans ce cas ;) Dans tous les cas je vous invite à parcourir la nouvelle version, qui a le mérite d'apporter plus de réponses qu'elle ne soulève d'interrogations. Ce qui n'était pas vraiment le cas avant, c'est donc un grand progrès :)

Accès plus facile Un certain nombre de changements ont été apportés à la librairie pour en rendre l'accès plus facile.

Notamment, comme on l'a évoqué en début de semaine, on peut maintenant faire fonctionner une simple coroutine à des fins pédagogiques en faisait plus simplement :

```
>>> import asyncio
>>> async def hello_world():
...     await asyncio.sleep(0.2)
...     print("hello world")
...
```

```
>>> asyncio.run(hello_world())
hello world
```

On a également créé des raccourcis, comme par exemple :

- `asyncio.create_task()` est un alias pour `asyncio.get_event_loop().create_task()` ;
- de même `asyncio.current_task()` et `asyncio.all_tasks()` font ce que vous imaginez ;

Commodité Changement un peu plus profond, la fonction `asyncio.get_running_loop()` permet d'accéder à la boucle courante.

Si vous avez lu du code `asyncio` plus ancien, vous avez peut-être remarqué une tendance prononcée à passer un objet loop en paramètre à peu près partout. Grâce à cette fonction, cela n'est plus nécessaire, on est garanti de pouvoir retrouver, à partir de n'importe quelle coroutine, l'objet boucle qui nous pilote.

De manière corollaire, une méthode `get_loop` a été ajoutée aux classes `Future` et `Task`.

Pas de changement de fond Sinon, en terme des concepts fondamentaux, tout le contenu du cours reste valide.

Pour en savoir plus VOus retrouverez tous les détails dans la page suivante :

https://docs.python.org/3/whatsnew/3.7.html#whatsnew37-asyncio

IPython7 et **`asyncio`**

`await` dans ipython-7

Cette section ne s'applique pas stricto sensu à Python-3.7, mais à la version 7 de IPython.

Le sujet, c'est ici encore de raccourcir le boilerplate nécessaire, lorsque vous avez écrit une coroutine et que vous voulez la tester.

Python standard

Voici d'abord ce qui se passe avec l'interpréteur Python standard (ici en 3.7) :

```
$ python3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
    <snip>
>>> import asyncio
>>>
>>> async def hello_world():
...     await asyncio.sleep(0.2)
...     print("hello world")
...
>>> await(hello_world())
  File "<stdin>", line 1
SyntaxError: 'await' outside function
>>>
>>> asyncio.run(hello_world())
hello world
```

La syntaxe de Python nous interdit en effet d'utiliser `await` en dehors du code d'une coroutine, on l'a vu dans une des vidéos, et il nous faut faire appel à `asyncio.run()`.

IPython-7 : on peut faire **await** au toplevel !

Pour simplifier encore la mise en place de code asynchrone, depuis ipython-7, on peut carrément déclencher une coroutine en invoquant `await` dans la boucle principale de l'interpréteur :

```
$ ipython3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import asyncio

In [2]: async def hello_world():
   ...:     await asyncio.sleep(0.2)
   ...:     print("hello world")

In [3]: await(hello_world())
hello world
```

Du coup, cette façon de faire fonctionnera aussi dans un notebook, si vous avez la bonne version de IPython en dessous de Jupyter.