

# Python 103



Programmation concurrente (3.6.4)



# Histoire de la concurrence

- La concurrence est en fait dérivée des premiers travaux sur les chemins de fer et la télégraphie, d'où l'utilisation de noms tels que le sémaphore
- Essentiellement, il était nécessaire de gérer plusieurs trains sur le même réseau ferroviaire de manière à ce que chaque train puisse se rendre à destination en toute sécurité sans encourir d'accident
- Ce n'est que dans les années 1960 que les chercheurs se sont intéressés à l'informatique concurrente, et c'est Edsger W. Dijkstra qui a publié le premier article dans ce domaine, où il a identifié et résolu le problème de l'exclusion mutuelle
- Ensuite, Dijkstra a défini des concepts fondamentaux de concurrence, tels que les sémaphores, les exclusions mutuelles et les blocages, ainsi que l'algorithme du plus court chemin de Dijkstra

# Histoire de la concurrence

- La concurrence, comme dans la plupart des domaines de l'informatique, est encore un domaine incroyablement jeune comparé à d'autres domaines d'étude tels que les mathématiques, et cela vaut la peine de garder cela à l'esprit
- Il y a encore un énorme potentiel de changement dans le domaine, et il reste un domaine passionnant pour tous (universitaires, concepteurs de langage et développeurs)
- L'introduction de primitives de concurrence de haut niveau et une meilleure prise en charge du langage natif ont vraiment amélioré la façon dont nous, en tant qu'architectes logiciels, mettons en œuvre des solutions concurrentes
- Pendant des années, c'était incroyablement difficile à faire, mais avec l'avènement de nouvelles API concurrentes, et la maturation des framework et des langages, cela commence à devenir beaucoup plus facile pour nous en tant que développeurs

# Histoire de la concurrence

- Les concepteurs de langages sont confrontés à un défi important lorsqu'ils tentent d'implémenter une concurrence non seulement sûre, mais efficace et facile à écrire pour les utilisateurs de cette langue
- Les langages de programmation tels que Golang, Rust et même Python ont fait de grands progrès dans ce domaine, et il est beaucoup plus facile d'exploiter tout le potentiel des machines sur lesquelles vos programmes fonctionnent

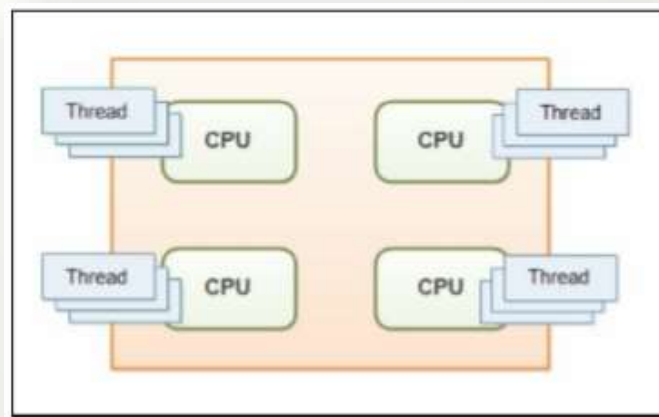
# Qu'est-ce qu'un thread ?

- Un thread peut être défini comme un flux ordonné d'instructions pouvant être programmé pour s'exécuter comme tel par les systèmes d'exploitation
- Ces threads, typiquement, vivent dans des processus, et consistent en un compteur de programme, une pile, et un ensemble de registres ainsi qu'un identifiant
- Ces threads sont la plus petite unité d'exécution à laquelle un processeur peut allouer du temps
- Les threads peuvent interagir avec des ressources partagées et la communication est possible entre plusieurs threads
- Ils sont également capables de partager de la mémoire, de lire et d'écrire différentes adresses de mémoire, mais c'est là un problème
- Lorsque deux threads commencent à partager de la mémoire et que vous n'avez aucun moyen de garantir l'ordre d'exécution d'un thread, vous pouvez commencer à voir des problèmes ou des bogues mineurs qui vous donnent des valeurs erronées ou font planter votre système



# Qu'est-ce qu'un thread ?

- La figure suivante montre comment plusieurs threads peuvent exister sur plusieurs processeurs différents



# Types de threads

- Dans un système d'exploitation typique, nous avons généralement deux types de threads distincts:
  - Threads au niveau utilisateur : Threads que nous pouvons activement créer, exécuter et tuer pour toutes nos tâches
  - Threads au niveau du noyau : Threads de très bas niveau au nom du système d'exploitation
- Python fonctionne au niveau de l'utilisateur, et donc, tout ce que nous couvrons dans ce document sera, principalement, axé sur ces threads au niveau de l'utilisateur

# Qu'est-ce que le multithreading ?

- Lorsque les gens parlent de processeurs multithread, ils font généralement référence à un processeur qui peut exécuter plusieurs threads simultanément, ce qu'ils sont capables de faire en utilisant un seul cœur capable de changer très rapidement le contexte entre plusieurs threads
- Ce contexte de commutation se déroule en si peu de temps que l'on pourrait penser que plusieurs threads fonctionnent en parallèle alors qu'en réalité ce n'est pas le cas
- Lorsque vous essayez de comprendre le multithreading, il est préférable de considérer un programme multithread comme un bureau
- Dans un programme monothread, il n'y aurait qu'une seule personne travaillant dans ce bureau à tout moment, manipulant tout le travail de manière séquentielle
- Cela deviendrait un problème si l'on considère ce qui se passe lorsque ce travailleur solitaire s'embourbe dans la paperasserie administrative et est incapable de passer à un travail différent



# Qu'est-ce que le multithreading ?

- Voyons quelques avantages du threading :
  - Les threads multiples sont excellents pour réduire le temps blocage des E/S liées aux programmes
  - Les threads sont légers en termes d'empreinte mémoire par rapport aux processus
  - Les threads partagent des ressources, et donc la communication entre eux est plus facile

# Qu'est-ce que le multithreading ?

- Il y a aussi des inconvénients, qui sont les suivants :
  - Les threads CPython sont entravés par les limitations du verrou global de l'interpréteur (GIL), dont nous parlerons plus en détail
  - Bien que la communication entre les threads puisse être plus facile, vous devez veiller à ne pas implémenter de code soumis à des conditions de concurrence
  - Il est coûteux en temps de changer de contexte entre plusieurs threads
  - En ajoutant plusieurs threads, vous pourriez voir une dégradation des performances globales de votre programme

# Qu'est-ce qu'un processus ?

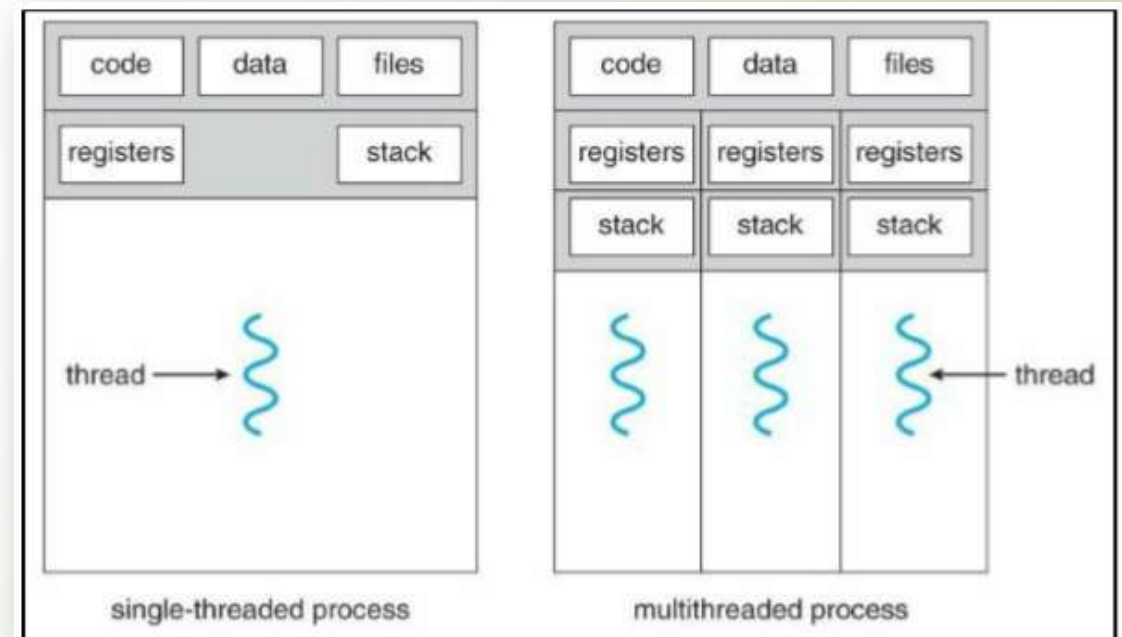
- Les processus sont très similaires aux threads
- Ils nous permettent de faire à peu près tout ce qu'un thread peut faire mais l'avantage principal est qu'ils ne sont pas liés à un noyau CPU singulier
- Si nous étendons notre analogie de bureau, cela signifie essentiellement que si nous avons un processeur à quatre cœurs, nous pourrions embaucher deux membres de l'équipe de vente dédiés et deux employés, et tous les quatre seraient en mesure d'exécuter le travail en parallèle
- Les processus sont également capables de travailler sur plusieurs choses à la fois, tout comme notre employé de bureau unique multithread

# Qu'est-ce qu'un processus ?

- Ces processus contiennent un thread principal primaire, mais peuvent engendrer plusieurs sous-threads qui contiennent chacun leur propre jeu de registres et une pile
- Ils peuvent devenir multithread si vous le souhaitez
- Tous les processus fournissent toutes les ressources dont l'ordinateur a besoin pour exécuter un programme

# Qu'est-ce qu'un processus ?

- Dans l'image suivante, vous voyez deux diagrammes côte-à-côte; les deux sont des exemples d'un processus
- Vous remarquez que le processus sur la gauche contient un seul thread, autrement connu comme le thread primaire
- Le processus sur la droite contient plusieurs threads, chacun avec son propre ensemble de registres et de piles





# Qu'est-ce qu'un processus ?

- Avec les processus, nous pouvons améliorer la vitesse de nos programmes dans des scénarios spécifiques où nos programmes sont liés au processeur et nécessitent plus de puissance CPU
- Cependant, en engendrant de multiples processus, nous sommes confrontés à de nouveaux défis en termes de communication interprocessus, et en tentant de ne pas entraver les performances en passant trop de temps sur cette communication interprocessus (IPC)

# Propriétés des processus ?

- Les processus UNIX sont créés par le système d'exploitation et contiennent généralement les éléments suivants
  - ID de processus
  - ID de groupe de processus
  - ID utilisateur et ID de groupe
  - Environnement
  - Répertoire de travail
  - Instructions de programme
  - Registres
  - Pile
  - Heap
  - Descripteurs de fichier
  - Actions de signal
  - Bibliothèques partagées
  - Outils de communication interprocessus (tels que files d'attente de messages, pipes, sémaphores ou mémoire partagée)

# Propriétés des processus ?

- Les avantages des processus sont les suivants :
  - Les processus peuvent mieux utiliser les processeurs multi cœurs
  - Ils sont meilleurs que les threads multiples pour gérer les tâches gourmandes en ressources
  - Nous pouvons contourner les limitations du GIL en engendrant plusieurs processus. Programme
  - Un processus qui se termine en erreur ne provoquera pas l'arrêt du programme entier

# Propriétés des processus ?

- Voici les principaux inconvénients des processus :
  - Pas de ressources partagées entre les processus
  - Nous devons mettre en œuvre une forme de IPC
  - Tout ceci nécessite plus de mémoire

# Multi processing ?

- En Python, nous pouvons choisir d'exécuter notre code à l'aide de plusieurs threads ou de plusieurs processus si nous souhaitons essayer d'améliorer les performances par rapport à une approche monothread standard
- Nous pouvons aller vers une approche multithread et être limité à la puissance de traitement d'un noyau de CPU, ou inversement,
- Nous pouvons aller avec une approche multi processing et utiliser le nombre total de cœurs de processeurs disponibles sur notre machine
- Dans les ordinateurs modernes d'aujourd'hui, nous avons tendance à avoir de nombreux processeurs et cœurs, de sorte que nous limiter à un seul, rend le reste de notre machine inactive
- Notre objectif est d'essayer d'extraire tout le potentiel de notre matériel et de nous assurer d'obtenir le meilleur rapport qualité-prix et de résoudre nos problèmes plus rapidement que quiconque



# Multi processing ?

- Avec le module multi processing de Python, nous pouvons utiliser efficacement le nombre total de cœurs et de processeurs, ce qui peut nous aider à obtenir de meilleures performances en ce qui concerne les problèmes liés au processeur



- La figure précédente montre un exemple de la façon dont un cœur CPU commence à déléguer des tâches à d'autres cœurs

# Multi processing ?

- Dans toutes les versions de Python inférieures ou égales à 2.6, nous pouvons obtenir le nombre de cœurs de processeur disponibles en utilisant le code suivant

```
>>> import multiprocessing
>>> multiprocessing.cpu_count()
8
>>>
```

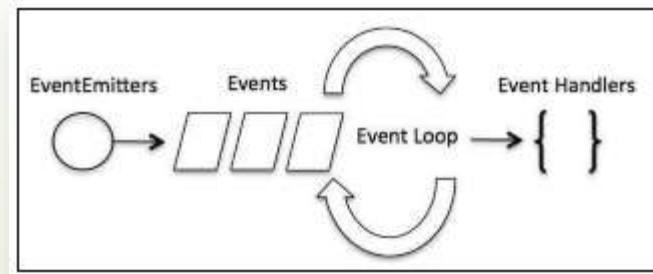
- Non seulement le multi processing nous permet d'utiliser plus de notre machine, mais nous évitons également les limitations que le verrou global d'interpréteur nous impose dans Python
- Un inconvénient potentiel de plusieurs processus est que nous n'avons intrinsèquement pas d'état partagé et que nous manquons de communication
- Nous devons donc passer à travers une forme de IPC, et la performance peut en prendre un coup
- Cependant, ce manque d'état partagé peut faciliter le travail, car vous n'avez pas à vous battre contre des conditions de concurrence potentielles dans votre code

# Programmation événementielle

- La programmation événementielle est une partie importante de nos vies
- Nous en voyons des exemples tous les jours lorsque nous ouvrons notre téléphone ou travaillons sur notre ordinateur
- Ces appareils fonctionnent uniquement de manière événementielle
- Par exemple, lorsque vous cliquez sur une icône sur votre bureau, le système d'exploitation enregistre cela comme un événement, puis effectue l'action nécessaire liée à ce style d'événement spécifique.

# Programmation événementielle

- Chaque interaction que nous faisons peut être caractérisée comme un événement ou une série d'événements, et ceux-ci déclenchent généralement des rappels (callbacks)
- Si vous avez une expérience antérieure avec JavaScript, vous devriez vous familiariser avec ce concept de rappel et le modèle de conception de rappel
- En JavaScript, le cas d'utilisation prédominant pour les rappels est lorsque vous effectuez des requêtes HTTP RESTful et que vous voulez pouvoir effectuer une action lorsque vous savez que cette action s'est terminée avec succès et que nous avons reçu notre réponse HTTP:



# Programmation événementielle

- Si nous regardons l'image précédente, elle nous montre un exemple de la façon dont les programmes pilotés par les événements traitent les événements
- Nous avons nos EventEmitters sur le côté gauche
- Ces derniers déclenchent plusieurs événements, qui sont captés par la boucle d'événements de notre programme, et, s'ils correspondent à un gestionnaire d'événements prédéfini, ce gestionnaire est alors déclenché pour gérer ledit événement



# Programmation événementielle

- Les rappels (callbacks) sont souvent utilisés dans les scénarios où une action est asynchrone
- Supposons, par exemple, que vous postuliez pour un emploi chez Google, que vous leur donniez une adresse e-mail, et qu'ils vous contacteront quand ils prendront leur décision
- Cela revient essentiellement à enregistrer un rappel sauf qu'au lieu de vous envoyer un e-mail, vous exécutez un code arbitraire chaque fois que le rappel est appelé

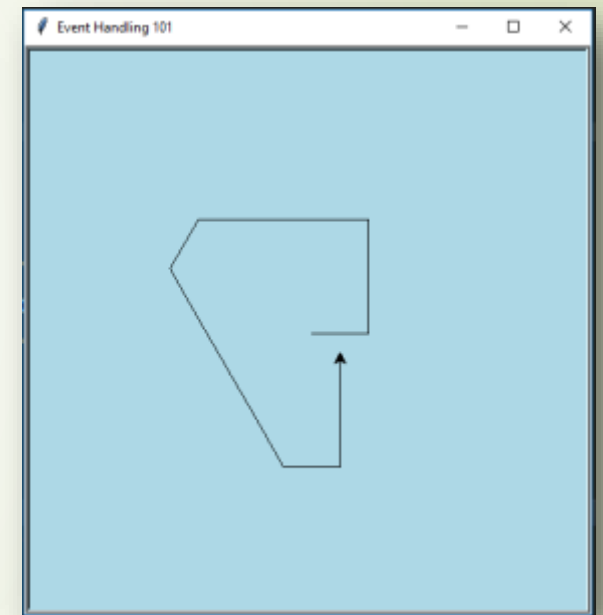
# Turtle

- Turtle est un module graphique qui a été écrit en Python, et c'est un excellent point de départ pour intéresser les enfants à la programmation
- Il gère toutes les complexités liées à la programmation graphique et leur permet de se concentrer uniquement sur l'apprentissage des bases tout en les gardant intéressés
- C'est aussi un très bon outil à utiliser pour démontrer des programmes axés sur les événements
- Il comporte des gestionnaires d'événements et des auditeurs, ce qui est tout ce dont nous avons besoin

```
turtle1.py x
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import turtle
5
6
7  def main():
8      import turtle
9      turtle.setup(500, 500)
10     window = turtle.Screen()
11     window.title("Event Handling 101")
12     window.bgcolor("lightblue")
13     nathan = turtle.Turtle()
14
15     def moveForward():
16         nathan.forward(50)
17
18     def moveLeft():
19         nathan.left(30)
20
21     def moveRight():
22         nathan.right(30)
23
24     def start():
25         window.onkey(moveForward, "Up")
26
27     window.onkey(moveForward, "Up")
28     window.onkey(moveLeft, "Left")
29     window.onkey(moveRight, "Right")
30     window.listen()
31     window.mainloop()
32
33
34 if __name__ == '__main__':
35     main()
36
```

# Turtle

- Dans la première ligne de cet exemple de code précédent, nous importons le module graphique Turtle
- Nous poursuivons ensuite pour mettre en place une fenêtre de tortue de base avec le titre Event Handling 101 et une couleur de fond de bleu clair
- Une fois la configuration initiale terminée, nous définissons trois gestionnaires d'événements distincts :
- `moveForward` : Quand nous voulons déplacer notre personnage de 50 unités
- `moveLeft/moveRight` : Quand nous voulons faire pivoter notre personnage dans les deux sens de 30 degrés
- Une fois que nous avons défini nos trois gestionnaires distincts, nous passons ensuite à la mise en correspondance de ces gestionnaires d'événements avec les touches vers le haut, vers la gauche et vers la droite en utilisant la méthode `onkey`



# Programmation réactive

- La programmation réactive est très similaire à celle de l'événement, mais au lieu de tourner autour des événements, elle se concentre sur les données
- Plus spécifiquement, elle traite des flux de données et réagit aux changements de données spécifiques

# ReactiveX - RxPy

- RxPy est l'équivalent Python du très populaire framework ReactiveX
- Si vous avez déjà effectué une programmation dans Angular 2 et versions suivantes, vous l'utiliserez lors de l'interaction avec les services HTTP
- Ce framework est une conglomération du modèle d'observateur, du modèle d'itérateur et de la programmation fonctionnelle
- Nous nous abonnons essentiellement à différents flux de données entrantes, puis créons des observateurs qui écoutent les événements spécifiques déclenchés
- Lorsque ces observateurs sont déclenchés, ils exécutent le code correspondant à ce qui vient de se passer



# ReactiveX - RxPy

- Nous considérerons un centre de données comme un bon exemple de la façon dont la programmation réactive peut être utilisée
- Imaginez que ce centre de données possède des milliers de racks de serveurs, qui calculent en permanence des millions et des millions de calculs
- L'un des plus grands défis de ces datacenters est de garder suffisamment refroidis tous ces racks de serveur bien serrés pour qu'ils ne s'endommagent pas
- Nous pourrions installer plusieurs thermomètres dans notre centre de données pour nous assurer de ne pas avoir trop chaud et envoyer les lectures de ces thermomètres à un ordinateur central en continu

# ReactiveX - RxPy

- Au sein de notre centrale de contrôle, nous pourrions mettre en place un programme RxPy qui observe ce flux continu d'informations sur la température
- Au sein de ces observateurs, nous pourrions alors définir une série d'événements conditionnels à écouter, puis réagir chaque fois que l'un de ces conditionnels est atteint
- Un tel exemple serait un événement qui se déclenche uniquement si la température d'une partie spécifique du centre de données devient trop chaude
- Lorsque cet événement est déclenché, nous pouvons alors automatiquement réagir et augmenter le débit de tout système de refroidissement vers cette zone particulière, et ainsi ramener la température à un bon niveau

# Programmation GPU

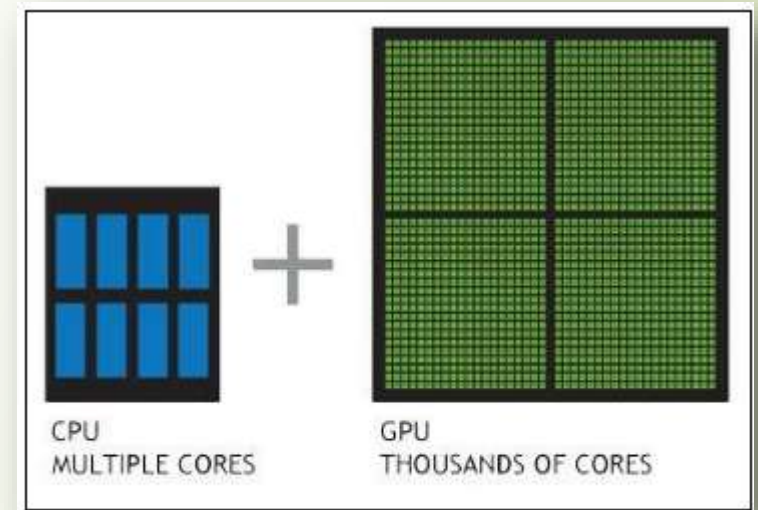
- Les GPU sont réputés pour leur capacité à rendre des jeux vidéo d'action rapides et de haute résolution
- Ils sont capables d'absorber ensemble les millions de calculs nécessaires par seconde afin de s'assurer que chaque vertex des modèles 3D de votre jeu est au bon endroit, et qu'ils sont mis à jour toutes les quelques millisecondes afin d'assurer une douceur de 60 FPS
- De manière générale, les GPU sont incroyablement performants pour effectuer la même tâche en parallèle, des millions et des millions de fois par minute
- Mais si les GPU sont si performants, alors pourquoi ne pas les utiliser à la place de nos processeurs ?
- Alors que les GPU peuvent être incroyablement performants au traitement graphique, ils ne sont cependant pas conçus pour gérer les subtilités de l'exécution d'un système d'exploitation et de l'informatique à usage général
- Les CPU ont moins de noyaux, qui sont spécifiquement conçus pour la vitesse quand il s'agit de changer de contexte entre les tâches d'exploitation
- Si les GPU avaient les mêmes tâches, vous constateriez une dégradation considérable des performances globales de votre ordinateur

# Programmation GPU

- Mais comment pouvons-nous utiliser ces cartes graphiques de haute puissance pour autre chose que la programmation graphique ?
- C'est ici qu'interviennent les bibliothèques telles que PyCUDA, OpenCL et Theano
- Ces bibliothèques tentent de faire abstraction du code complexe de bas niveau auquel les API graphiques doivent interagir pour utiliser le GPU
- Ils simplifient grandement la réutilisation des milliers de cœurs de traitement plus petits disponibles sur le GPU et leur utilisation pour nos programmes coûteux en calcul

# Programmation GPU

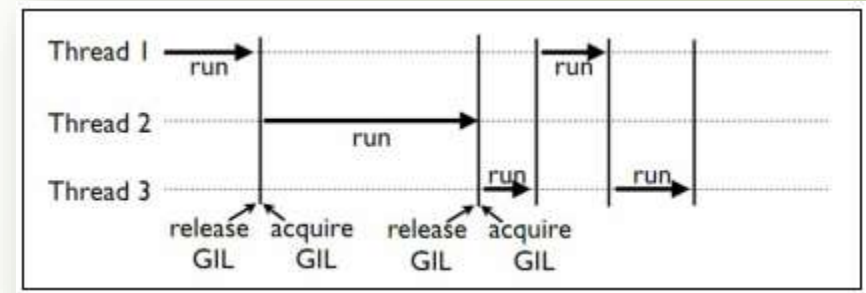
- Ces unités de traitement graphique (GPU) encapsulent tout ce que les langages de script ne sont pas
- Ils sont hautement parallélisables et construits pour un débit maximum
- En les utilisant en Python, nous sommes en mesure d'obtenir le meilleur des deux mondes
- Nous pouvons utiliser un langage favorisé par des millions de personnes en raison de sa facilité d'utilisation, et rendre nos programmes incroyablement performants





# Les limitations de Python

- Plus tôt dans le chapitre, j'ai parlé des limites du GIL ou du Global Interpreter Lock présent dans Python, mais qu'est-ce que cela signifie réellement ?
- Premièrement, je pense qu'il est important de savoir exactement ce que le GIL fait pour nous
- Le GIL est essentiellement un verrou d'exclusion mutuelle qui empêche plusieurs threads d'exécuter du code Python en parallèle
- C'est un verrou qui ne peut être tenu que par un thread à la fois, et si vous voulez qu'un thread exécute son propre code, il doit d'abord acquérir le verrou avant de pouvoir exécuter son propre code
- L'avantage que cela nous procure est que pendant qu'il est verrouillé, rien d'autre ne peut fonctionner en même temps



# Les limitations de Python

- Dans le diagramme précédent, nous voyons un exemple de la façon dont plusieurs threads sont entravés par ce GIL
- Chaque thread doit attendre et acquérir le GIL avant de pouvoir continuer à progresser, puis relâcher le GIL, généralement avant qu'il n'ait pu terminer son travail
- Il suit une approche aléatoire à la ronde, et vous n'avez aucune garantie quant au thread qui va acquérir le verrou en premier
- Pourquoi est-ce nécessaire, vous pourriez demander?
- Eh bien, le GIL a longtemps été une partie de Python, et au fil des années a déclenché de nombreux débats sur son utilité
- Mais il a été implémenté avec de bonnes intentions et pour lutter contre la gestion de la mémoire Python non-thread-safe
- Il nous empêche de tirer parti des systèmes multiprocesseurs dans certains scénarios

# Les limitations de Python

- Guido Van Rossum, le créateur de Python, a publié une mise à jour sur la suppression du GIL et de ses avantages dans un post ici :
- <http://www.artima.com/weblogs/viewpost.jsp?thread=214235>
- Il déclare qu'il ne serait pas contre quelqu'un créant une branche de Python qui soit sans GIL, et il accepterait une fusion de ce code si, et seulement si, cela n'avait pas d'impact négatif sur la performance d'une application mono threaded
- Il y a eu des tentatives antérieures pour se débarrasser du GIL, mais il a été constaté que l'ajout de tous les verrous supplémentaires pour assurer la sécurité du fil ralentissait réellement une application d'un facteur de plus de deux
- En d'autres termes, vous auriez pu faire plus de travail avec un seul processeur qu'avec un peu plus de deux processeurs
- Il y a cependant des bibliothèques comme NumPy qui peuvent faire tout ce dont elles ont besoin sans avoir à interagir avec le GIL, et travailler en dehors du GIL est quelque chose que nous approfondirons dans les prochains chapitres de ce livre

# Les limitations de Python

- Il faut également noter qu'il existe d'autres implémentations de Python, telles que Jython et IronPython, qui n'offrent aucune forme de verrouillage d'interpréteur global et, en tant que telles, peuvent exploiter pleinement les systèmes multiprocesseurs
- Jython et IronPython s'exécutent sur des machines virtuelles différentes, ce qui leur permet de tirer parti de leurs environnements d'exécution respectifs

# Jython

- Jython est une implémentation de Python qui fonctionne directement avec la plateforme Java
- Il peut être utilisé de manière complémentaire avec Java en tant que langage de script, et il a été démontré qu'il surpassait CPython, qui est l'implémentation standard de Python, lorsqu'il travaille avec de grands ensembles de données
- Cependant, pour la majorité des choses, l'exécution par un seul noyau de CPython surpasse généralement Jython et son approche multi cœur
- L'avantage de l'utilisation de Jython réside dans le fait que vous pouvez faire des choses très intéressantes lorsque vous travaillez en Java, comme importer des bibliothèques et des framework Java existants, et les utiliser comme s'ils faisaient partie de votre code Python.



# IronPython

- IronPython est l'équivalent .NET de Jython et fonctionne au-dessus du framework .NET de Microsoft
- Encore une fois, vous serez en mesure de l'utiliser de manière complémentaire avec les applications .NET
- Ceci est quelque peu bénéfique pour les développeurs .NET, car ils sont capables d'utiliser Python comme langage de script rapide et expressif dans leurs applications .NET

# Pourquoi utiliser Python ?

- Si Python a des limitations évidentes et connues quand il s'agit d'écrire des applications performantes, alors pourquoi continuons-nous à l'utiliser ?
- La réponse courte est que c'est un langage fantastique pour faire le travail, et par le travail, je ne parle pas nécessairement d'une tâche coûteuse en termes de calcul
- C'est un langage intuitif, facile à comprendre et à apprendre pour ceux qui n'ont pas forcément beaucoup d'expérience en programmation
- Le langage a vu un taux d'adoption énorme parmi les scientifiques des données et les mathématiciens travaillant dans des domaines incroyablement intéressants tels que l'apprentissage automatique et l'analyse quantitative, qui trouvent que c'est un outil incroyablement utile dans leur arsenal
- Dans les deux écosystèmes Python 2 et 3, vous trouverez un grand nombre de bibliothèques conçues spécifiquement pour ces cas d'utilisation, et en connaissant les limites de Python, nous pouvons les atténuer efficacement, et produire un logiciel efficace et capable de faire exactement ce qui est requis

# Téléchargement d'images

- Un excellent exemple des avantages du multithreading est, sans aucun doute, l'utilisation de plusieurs threads pour télécharger plusieurs images ou fichiers
- C'est, en fait, l'un des meilleurs cas d'utilisation pour le multithreading en raison de la nature bloquante des E/S
- Pour mettre en évidence les gains de performance, nous allons récupérer 10 images différentes de <http://lorempixel.com/400/200/sports>, qui est une API gratuite qui fournit une image différente chaque fois que vous cliquez sur ce lien
- Nous allons ensuite stocker ces 10 images différentes dans un dossier temporaire afin que nous puissions les voir ou les utiliser plus tard

# Téléchargement d'images : *Séquentiel*

- Premièrement, nous devrions disposer d'une base de référence permettant de mesurer les gains de performance
- Pour ce faire, nous écrirons un programme rapide qui téléchargera séquentiellement ces 10 images, comme suit :

```
concurrency1.py x
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import time
5  import urllib.request
6
7
8  def downloadImage(imagePath, fileName):
9      print("Downloading Image from ", imagePath)
10     urllib.request.urlretrieve(imagePath, fileName)
11
12
13  def main():
14      t0 = time.time()
15      for i in range(10):
16          imageName = "temp/image-" + str(i) + ".jpg"
17          downloadImage("http://lorempixel.com/400/200/sports", imageName)
18          t1 = time.time()
19          total_time = t1 - t0
20      print("Execution time: {}".format(total_time))
21
22
23  if __name__ == '__main__':
24      main()
25
```

```
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Execution time: 2.46917724609375
```



# Téléchargement d'images : *Concurrent*

- Maintenant que nous avons notre base de référence, il est temps d'écrire un programme rapide qui téléchargera simultanément toutes les images dont nous avons besoin
- Nous allons continuer à créer et à démarrer des threads dans les prochains chapitres, donc ne vous inquiétez pas si vous avez du mal à comprendre le code
- Le point clé de ceci est de réaliser les gains de performance potentiels à avoir en écrivant des programmes simultanément

```
concurrency2.py x
1  #!/usr/bin/env python
2  #-*- coding: utf-8 -*-
3
4
5  import threading, urllib.request, time
6
7
8  def downloadImage(imagePath, fileName):
9      print("Downloading Image from ", imagePath)
10     urllib.request.urlretrieve(imagePath, fileName)
11
12
13  def executeThread(i):
14      imageName = "temp/image-" + str(i) + ".jpg"
15      downloadImage("http://lorempixel.com/400/200/sports", imageName)
16
17
18  def main():
19      t0 = time.time()
20      threads = []
21      for i in range(10):
22          thread = threading.Thread(target=executeThread, args=(i,))
23          threads.append(thread)
24          thread.start()
25      for i in threads:
26          i.join()
27      t1 = time.time()
28      totalTime = t1 - t0
29      print("Total Execution Time {}".format(totalTime))
30
31
32  if __name__ == '__main__':
33      main()
34
```

```
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Downloading Image from http://lorempixel.com/400/200/sports
Total Execution Time 0.5017025470733643
```



# Améliorer le calcul

- Ainsi, nous avons vu exactement comment nous pouvons améliorer des choses telles que le téléchargement d'images, mais comment pouvons-nous améliorer la performance de notre calcul de nombres ?
- Eh bien, c'est là que le multitraitement brille s'il est utilisé correctement
- Dans cet exemple, nous allons essayer de trouver les facteurs premiers de 10 000 nombres aléatoires compris entre 20 000 et 100 000 000
- Nous ne sommes pas nécessairement préoccupés par l'ordre d'exécution tant que le travail est effectué et que nous ne partageons pas la mémoire entre nos processus

# Factorisation séquentielle

- Encore une fois, nous allons écrire un script qui le fait d'une manière séquentielle, que nous pouvons facilement vérifier fonctionne correctement :

```

9991 37016767 -> [73, 507079]
9992 15042428 -> [2, 2, 811, 4637]
9993 66177718 -> [2, 3623, 9133]
9994 59639200 -> [2, 2, 2, 2, 2, 5, 5, 127, 587]
9995 10271140 -> [2, 2, 5, 11, 46687]
9996 30917369 -> [7, 643, 6869]
9997 87089 -> [73, 1193]
9998 77469870 -> [2, 3, 5, 2582329]
9999 41286402 -> [2, 3, 3, 3, 764563]
Execution Time: 1.750622272491455

```

```

concurrency3.py
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  import time
6  import random
7
8
9  def calculate_prime_factors(n):
10     primfac = []
11     d = 2
12     while d * d <= n:
13         while (n % d) == 0:
14             primfac.append(d)
15             n //= d
16         d += 1
17     if n > 1:
18         primfac.append(n)
19     return primfac
20
21
22 def main():
23     print("Starting number crunching")
24     t0 = time.time()
25     for i in range(10000):
26         rand = random.randint(20000, 100000000)
27         print(i, rand, '| -> ', calculate_prime_factors(rand))
28     t1 = time.time()
29     total_time = t1 - t0
30     print("Execution Time: {}".format(total_time))
31
32
33 if __name__ == '__main__':
34     main()
35

```

# Factorisation concurrente

- Voyons maintenant comment nous pouvons améliorer les performances de ce programme en utilisant plusieurs processus
- Afin de diviser cette charge de travail, nous allons définir une fonction **executeProc** qui, au lieu de générer 10 000 nombres aléatoires à factoriser, générera 1 000 nombres aléatoires
- Nous allons créer 10 processus, et exécuter la fonction 10 fois, de sorte que le nombre total de calculs devrait être exactement le même que lorsque nous avons effectué le test séquentiel

```
29020301 -> [29020301]
95560538 -> [2, 19, 23, 31, 3527]
8703773 -> [13, 607, 1103]
65542457 -> [19, 3449603]
22495699 -> [401, 56099]
Execution Time: 1.3285346031188965
```

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import time
5 import random
6 from multiprocessing import Process
7
8
9 def calculatePrimeFactors(n):
10     primfac = []
11     d = 2
12     while d * d <= n:
13         while (n % d) == 0:
14             primfac.append(d)
15             n //= d
16         d += 1
17     if n > 1:
18         primfac.append(n)
19     return primfac
20
21
22 def executeProc():
23     for i in range(1000):
24         rand = random.randint(20000, 100000000)
25         print(rand, '->', calculatePrimeFactors(rand))
26
27
28 def main():
29     print("Starting number crunching")
30     t0 = time.time()
31     procs = []
32     for i in range(10):
33         proc = Process(target=executeProc, args=())
34         procs.append(proc)
35         proc.start()
36     for proc in procs:
37         proc.join()
38     t1 = time.time()
39     totalTime = t1 - t0
40     print("Execution Time: {}".format(totalTime))
41
42
43 if __name__ == '__main__':
44     main()
45
```

# Concurrence vs Parallélisme

- La concurrence et le parallélisme sont deux concepts généralement confondus
- La réalité, cependant, est qu'ils sont très différents, et si vous avez conçu un logiciel concurrent au lieu d'avoir besoin d'une exécution parallèle, alors vous pourriez sérieusement affecter le véritable potentiel de performance de votre logiciel
- Pour cette raison, il est essentiel de savoir exactement ce que les deux concepts signifient afin que vous puissiez comprendre les différences
- En connaissant ces différences, vous aurez les clés pour concevoir vos propres logiciels haute performance en Python



# Comprendre la concurrence

- La concurrence est, essentiellement, de faire plusieurs choses en même temps, mais pas, spécifiquement, en parallèle
- Cela peut nous aider à améliorer les performances perçues de nos applications, et cela peut également améliorer la vitesse à laquelle nos applications fonctionnent
- La meilleure façon de voir comment la concurrence fonctionne est d'imaginer une personne travaillant sur plusieurs tâches et passant rapidement d'une tâche à l'autre
- Imaginez que cette personne travaille simultanément sur un programme et, en même temps, traite des demandes de support
- Cette personne se concentrerait principalement sur l'écriture de son programme, et rapidement le contexte passerait à la résolution d'un bug ou à la gestion d'un problème de support s'il y en avait un
- Une fois qu'ils ont terminé la tâche de support, ils peuvent à nouveau changer de contexte, pour écrire leur programme très rapidement



# Comprendre la concurrence

- Cependant, en informatique, il existe généralement deux goulets d'étranglement des performances que nous devons surveiller et éviter en écrivant nos programmes
- Il est important de connaître les différences entre les deux goulets d'étranglement
- Si vous essayez d'appliquer la concurrence à un goulot d'étranglement basé sur le processeur, alors vous constaterez que le programme commence à voir une diminution des performances
- Et si vous avez essayé d'appliquer le parallélisme à une tâche qui nécessite réellement une solution concurrente, vous pouvez à nouveau voir les mêmes performances

# Propriétés d'un système concurrent

- Tous les systèmes concurrents partagent un ensemble de propriétés similaire
- Ceux-ci peuvent être définis comme suit :
- Acteurs multiples : Ceci représente les différents processus et threads qui essaient tous de progresser activement sur leurs propres tâches. Nous pourrions avoir plusieurs processus qui contiennent plusieurs threads essayant tous de s'exécuter en même temps
- Ressources partagées : Cette fonctionnalité représente la mémoire, le disque et les autres ressources que les acteurs du groupe précédent doivent utiliser pour effectuer ce qu'ils doivent faire
- Règles : Il s'agit d'un ensemble strict de règles que tous les systèmes concurrents doivent suivre et qui définissent quand les acteurs peuvent et ne peuvent pas acquérir de verrous, accéder à la mémoire, modifier l'état, etc. Ces règles sont essentielles pour que ces systèmes concurrents fonctionnent, sinon, nos programmes se déchireraient les uns les autres

# Goulots d'étranglement d'E/S

- Les goulots d'étranglement d'E/S sont des goulots d'étranglement où votre ordinateur passe plus de temps à attendre diverses entrées et sorties qu'à traiter les informations
- Vous trouverez généralement ce type de goulot d'étranglement lorsque vous travaillez avec une application lourde en E/S
- Nous pourrions considérer votre navigateur Web standard comme un exemple d'application aux E/S lourdes
- Dans un navigateur, nous passons généralement beaucoup plus de temps à attendre que les requêtes réseau se terminent pour charger des feuilles de style, des scripts ou des pages HTML plutôt que de les afficher à l'écran
- Si la vitesse à laquelle les données sont demandées est plus lente que la vitesse à laquelle elles sont consommées, alors vous avez un goulot d'étranglement d'E/S

# Goulots d'étranglement d'E/S

- L'un des principaux moyens d'améliorer la vitesse de ces applications est soit d'améliorer la vitesse des E/S sous-jacentes en achetant du matériel plus coûteux et plus rapide, soit d'améliorer la façon dont nous traitons ces demandes d'E/S
- Un excellent exemple de programme lié par les goulots d'étranglement d'E/S serait un robot d'indexation Web
- L'objectif principal d'un robot d'exploration Web est désormais de parcourir le Web et d'indexer les pages Web pour qu'elles puissent être prises en compte lorsque Google exécute son algorithme de classement des résultats pour déterminer les 10 premiers résultats d'un mot clé donné

# Goulots d'étranglement d'E/S

- Nous allons commencer par créer un script très simple qui ne demande qu'une page et combien de temps il faut pour demander cette page web

```
concurrency5.py x
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import urllib.request
5  import time
6
7
8  def main():
9      t0 = time.time()
10     request = urllib.request.urlopen("http://www.iana.org/domains/reserved")
11     page_html = request.read()
12     t1 = time.time()
13     print("\n\nTotal Time To Fetch Page: {} Seconds\n\n".format(t1 - t0))
14
15
16  if __name__ == '__main__':
17      #
18      # Call the main() function
19      #
20      main()
21
```

```
Total Time To Fetch Page: 0.562730073928833 Seconds
```



# Goulots d'étranglement d'E/S

- Maintenant, disons que nous voulions ajouter un peu de complexité et suivre les liens vers d'autres pages afin que nous puissions les indexer dans le futur
- Nous pourrions utiliser une bibliothèque comme **BeautifulSoup** afin de rendre nos vies un peu plus faciles
- Vous remarquerez à partir de cette sortie que le temps d'aller chercher la page est supérieur à une demie seconde
- Maintenant, imaginons que si nous voulions exécuter notre robot d'exploration pour un million de pages Web différentes, notre temps d'exécution total serait environ un million de fois plus long...

```
concurrency6.py x
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  import urllib.request
6  import time
7  from bs4 import BeautifulSoup
8
9
10 def main():
11     t0 = time.time()
12     req = urllib.request.urlopen("http://www.iana.org/domains/reserved")
13     t1 = time.time()
14     print("\n\nTotal Time To Fetch Page: {} Seconds".format(t1 - t0))
15     soup = BeautifulSoup(req.read(), "html.parser")
16     for link in soup.find_all('a'):
17         link.get('href')
18     t2 = time.time()
19     print("Total Exececution Time: {} Seconds\n\n".format(t2 - t0))
20
21
22 if __name__ == '__main__':
23     main()
24
25
```

```
Total Time To Fetch Page: 0.5942203998565674 Seconds
Total Exececution Time: 0.6097383499145508 Seconds
```

# Goulots d'étranglement d'E/S

- La vraie cause principale de ce temps d'exécution énorme résulterait purement et simplement du goulot d'étranglement des E/S auquel nous sommes confrontés dans notre programme
- Nous passons énormément de temps à attendre sur nos demandes réseau, et une fraction de ce temps à analyser notre page récupérée pour d'autres liens à explorer

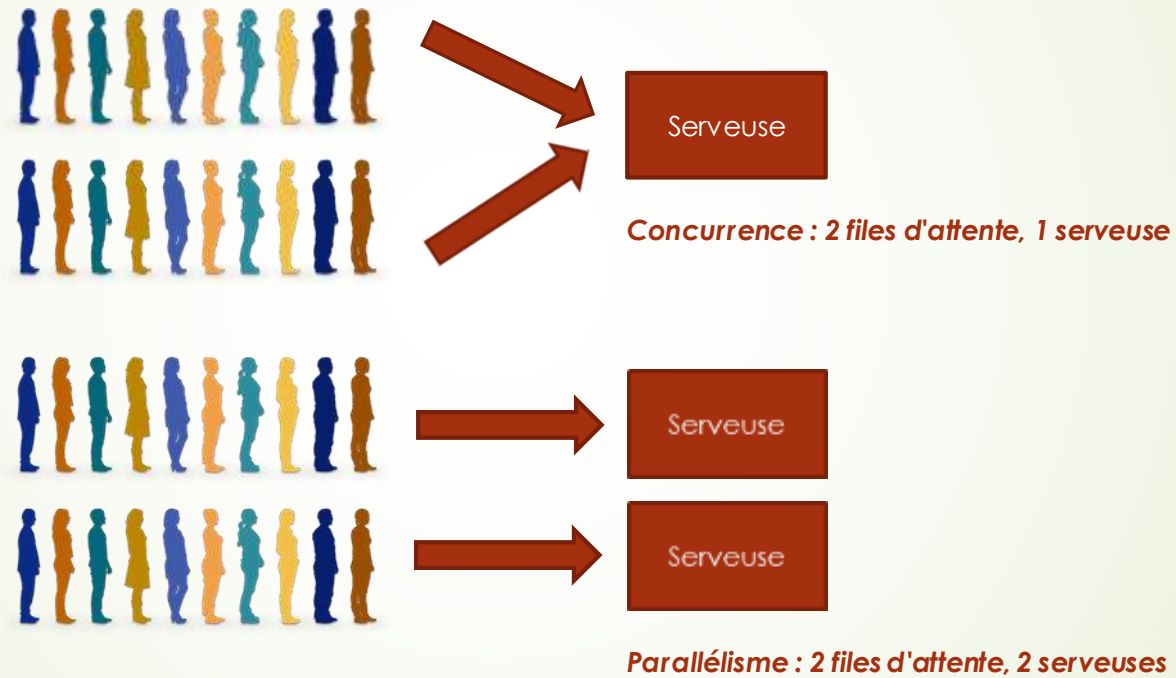
# Comprendre le parallélisme

- Au début, nous avons abordé un peu les capacités de multitraitement de Python, et comment nous pourrions l'utiliser pour tirer parti d'un plus grand nombre de cœurs de traitement dans notre matériel
- Mais que voulons-nous dire lorsque nous disons que nos programmes fonctionnent en parallèle ?
- Le parallélisme est l'art d'exécuter deux ou plusieurs actions simultanément, par opposition à la simultanéité dans laquelle vous faites des progrès sur deux ou plusieurs choses en même temps
- C'est une distinction importante, et afin d'obtenir un véritable parallélisme, nous aurons besoin de plusieurs processeurs pour exécuter nos codes en même temps

# Comprendre le parallélisme

- Une bonne analogie pour le traitement parallèle est de penser à une file d'attente pour le pain
- Si vous avez, par exemple, deux files d'attente de 20 personnes, toutes en attente d'avoir une baguette servie par la boulangère, ce serait un exemple de concurrence
- Maintenant, ajoutez une deuxième serveuse dans la boulangerie, ce serait alors un exemple de quelque chose qui se passe en parallèle
- C'est exactement comme cela que fonctionne le traitement en parallèle - chacune des serveuses dans cette boulangerie représente un noyau de traitement, et est capable de faire des progrès sur les tâches simultanément

# Comprendre le parallélisme





# Comprendre le parallélisme

- Un exemple concret qui met en évidence la véritable puissance du traitement parallèle est la carte graphique de votre ordinateur
- Ces cartes graphiques ont tendance à avoir des centaines, voire des milliers, de cœurs de traitement individuels qui vivent indépendamment, et peuvent calculer des choses en même temps
- La raison pour laquelle nous sommes en mesure d'exécuter des jeux PC haut de gamme à de telles cadences est due au fait que nous avons pu mettre autant de cœurs parallèles sur ces cartes

# Références

- Python.org : <https://www.python.org/>
- Learning Python : <https://github.com/thierrydecker/learning-python>
- ...

# Outils

- IDE Pycharm Community : <https://www.jetbrains.com/pycharm/>
- Analyse en ligne de code Python : <http://www.pythontutor.com/>
- ...