EPFL

ICC
Programmation

SIE/GCG, **Cours 12**

13 mai 2024

Jean-Philippe Pellet

Previously, on ICC/Programmation…

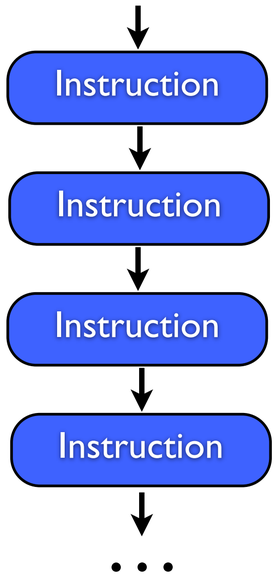
- Types de base en Python: int, float, str, bool
- Méthodes, fonctions et slicing pour calculer des valeurs dérivées
- Conditions pour exécuter du code selon la valeur d'une expression booléenne
- Boucles pour exécuter du code plusieurs fois:
- Déclaration de fonctions avec type de retour et paramètres
- Utilisation de fonctions comme valeurs et de fonctions d'ordre supérieur
- Utilisation de listes, sets et dictionnaires
- Utilisation de compréhensions de listes
- Déclaration de classes: `@dataclass class Rectangle: ...`
- Création, chargement, manipulation et sauvegarde d'images
- Programmation dynamique pour trouver des seams
- Lecture et écriture de fichiers texte

Cours de cette semaine

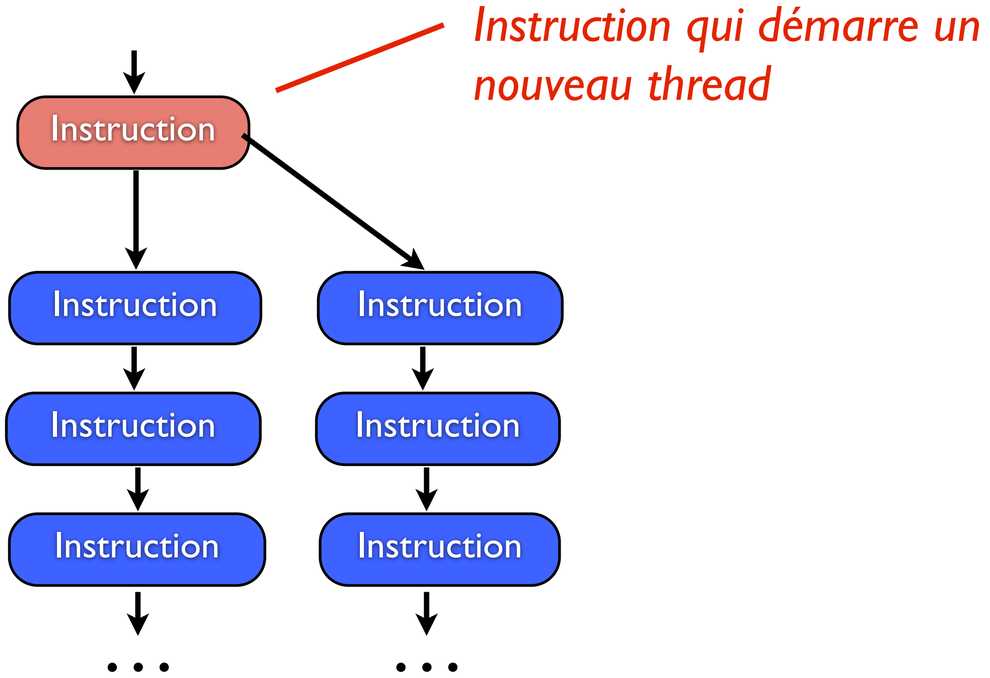
Threads
Problèmes des données partagées

Le *thread*: un fil d'exécution

Un seul thread
(ce qu'on connaît)



Plusieurs threads



Démo

Les threads en Python

```
from threading import Thread, current_thread
from time import sleep

def say_hello_periodically() -> None:
    thread_name = current_thread().getName()
    while True:
        print(f"Hello from thread {thread_name}")
        sleep(1.0)

thread1 = Thread(target=say_hello_periodically)
thread2 = Thread(target=say_hello_periodically)

thread1.start()
thread2.start()
```

On importe les choses nécessaires

Cette fonction sera indiquée aux nouveaux threads pour exécution

On dit bonjour en affichant le nom du thread

On attend une seconde avant de répéter la boucle

Deux nouveaux threads avec comme tâche à faire le code préparé en haut

On démarre enfin les threads.
L'exécution de la ligne suivante commence immédiatement!

Threads

- Un Thread est un «fil d'exécution»
  - Construit avec une fonction à exécuter pour lui dire quoi faire
    - ➡ Plusieurs threads peuvent utiliser la même fonction, comme dans l'exemple d'avant
  - On n'appelle pas la fonction avec `()`, mais on indique juste son nom
  - On le démarre en appelant sa méthode `start()`
  - La ligne suivante est exécutée ensuite tout de suite, sans attendre que la fonction donnée au thread démarré ait fini de s'exécuter

Exécution des threads

- Le code des threads s'exécute en parallèle — ou de manière concurrente
  - Vraiment parallèle si plusieurs processeurs
  - Sinon, un petit bout de chaque thread est exécuté, puis on change (système *round-robin*)
- Le système d'exploitation s'occupe de la planification (*scheduling*) des threads
  - En fonction des autres programmes qui tournent
  - Impossible de prédire ce qui va être exécuté quand entre plusieurs threads!
  - Problème pour l'accès aux données partagées entre plusieurs threads

Exemple: retrait et dépôt d'argent

```python
class BankAccount:
    def __init__(self) -> None:
        self.balance: int = 0

    def change_balance(self, delta: int) -> None:
        new_balance = self.balance + delta
        self.balance = new_balance
```

Situation initiale: balance == 1000

Vous déposez 200 fr. sur un thread du système de transactions bancaires

```
account.change_balance(delta=200)
```

```
new_balance = self.balance + 200
self.balance = new_balance
```

Votre partenaire retire 100 fr. «sur un autre thread»

```
account.change_balance(delta=-100)
```

```
new_balance = self.balance - 100
self.balance = new_balance
```

Résultat: balance == ? **1100, 900, 1200!**

Les scénarios possibles

Scénario 1

```
# balance vaut 1000

new_balance = self.balance + 200
# new_balance du thread 1 vaut 1200
self.balance = new_balance
# balance vaut 1200

-- CHANGEMENT DE THREAD

    new_balance = self.balance - 100
    # new_balance du thread 2 vaut 1100
    self.balance = new_balance
    # balance vaut 1100
```

Résultat: 1100

Scénario 2

```
# balance vaut 1000

new_balance = self.balance + 200
# new_balance du thread 1 vaut 1200

-- CHANGEMENT DE THREAD

    new_balance = self.balance - 100
    # new_balance du thread 2 vaut 900
    self.balance = new_balance
    # balance vaut 900

-- CHANGEMENT DE THREAD

self.balance = new_balance
# balance vaut 1200!
```

Résultat: 1200! :-)

Les scénarios possibles *(suite)*

Scénario 3

```
# balance vaut 1000

new_balance = self.balance + 200
# new_balance du thread 1 vaut 1200

-- CHANGEMENT DE THREAD

    new_balance = self.balance - 100
    # new_balance du thread 2 vaut 900

-- CHANGEMENT DE THREAD

self.balance = new_balance
# balance vaut 1200!

-- CHANGEMENT DE THREAD

    self.balance = new_balance
    # balance vaut 900!
```

Beaucoup d'autres scénarios possibles, surtout avec davantage de threads

Résultat: 900! :-(

Locks

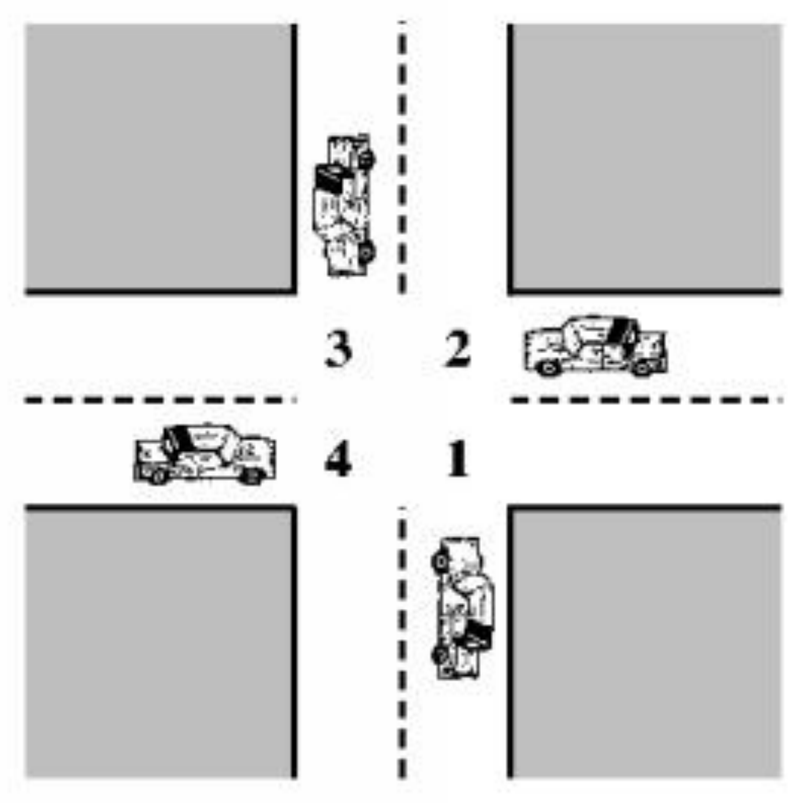
```
from threading import Lock

class BankAccount:
    def __init__(self) -> None:
        self.balance: int = 0
        self.lock = Lock()

    def change_balance(self, delta: int) -> None:
        with self.lock:
            new_balance = self.balance + delta
            self.balance = new_balance
```

Un *Lock* (verrou) est créé pour «fermer à clé» pendant qu'on travaille sur cet objet

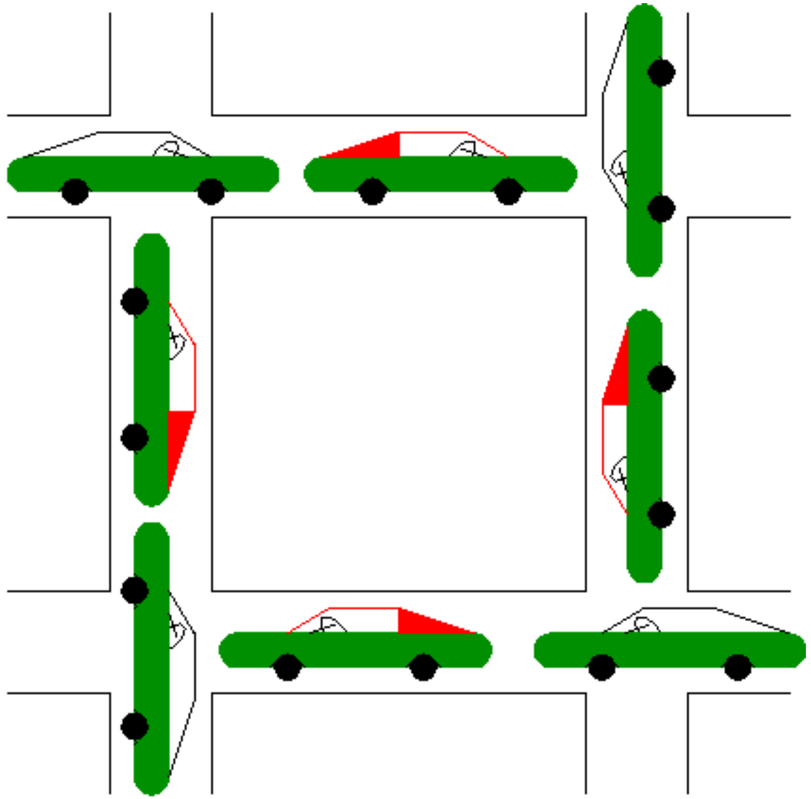
Ceci se passe uniquement avec «la porte verrouillée»

- Maintenant, le seul scénario possible est celui où le premier thread qui appelle cette méthode finit complètement la méthode avant qu'un autre y entre
- Les autres threads sont bloqués et doivent attendre
- OK dans ce cas-ci, mais rapidement complexe dans du code réel: plusieurs threads peuvent s'attendre les uns les autres et être tous bloqués, c'est un deadlock

Deadlocks dans la vie de tous les jours



Qui a la priorité?



Qui a une assez bonne vue d'ensemble pour résoudre ce problème?

Threads et interfaces graphiques

- Dans toute application graphique:
  - Le thread principal est utilisé par l'interface graphique pour «attendre» et réagir aux événements (clavier, souris, etc.)
  - Pour toute opération qui prend longtemps, on crée un autre thread
    - ➡ Ceci permet à l'interface graphique de continuer à réagir
    - ➡ Par exemple: un bouton «stop» pour arrêter une opération en cours
- Exemple des pages web
  - Le code JavaScript a thread principal qui bloque les interactions avec la page
  - On a le droit de créer de nouveaux threads spéciaux, des *web workers*
  - Les *web workers* n'ont (presque) pas le droit de partager des données avec le thread principal

Résumé Cours 12

- Les threads servent à exécuter plusieurs morceaux de code «à la fois»
- Il peut y avoir des problèmes lorsque ces threads doivent changer des variables communes
- Les locks peuvent servir à rendre l'exécution d'une partie du code séquentielle, mais ne résolvent pas tous les problèmes
- Un deadlock peut survenir si plusieurs threads s'attendent les uns les autres

- *Séance d'exercice: examens des années passées*