

## Information, Calcul et Communication

### CS-119(k) ICC – Programmation Semaine 12

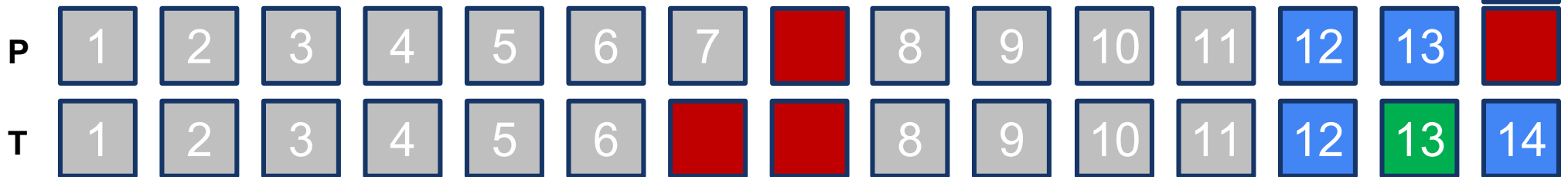
Rafael Pires

[rafael.pires@epfl.ch](mailto:rafael.pires@epfl.ch)

# Planning

11.05 **Vous êtes là**

Cours et séries, partie programmation

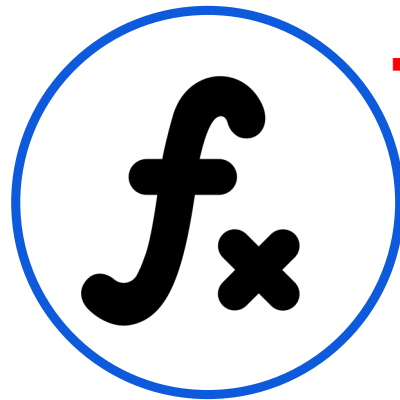
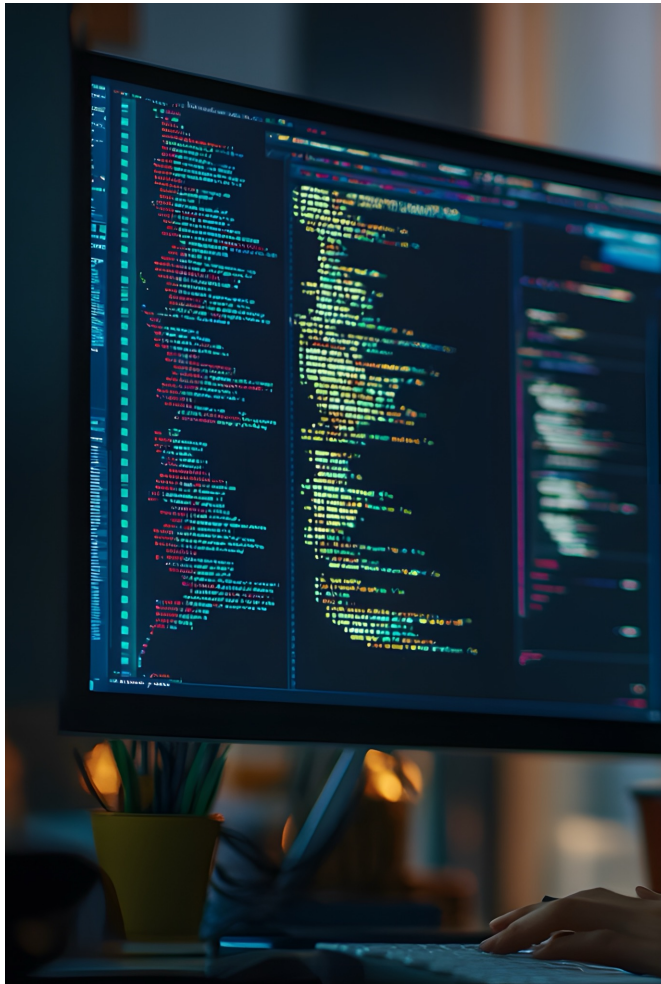


Cours et séries, partie théorique

23.05 Miniprojet P  
(10%)

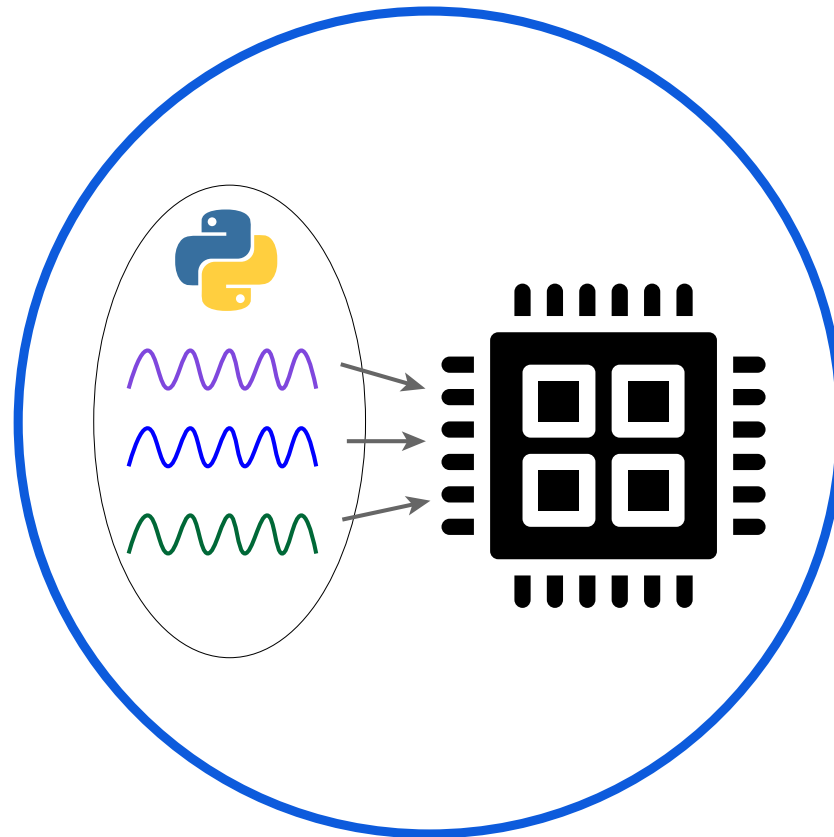
25.05 Congé

# Précédemment, dans... ICC-P

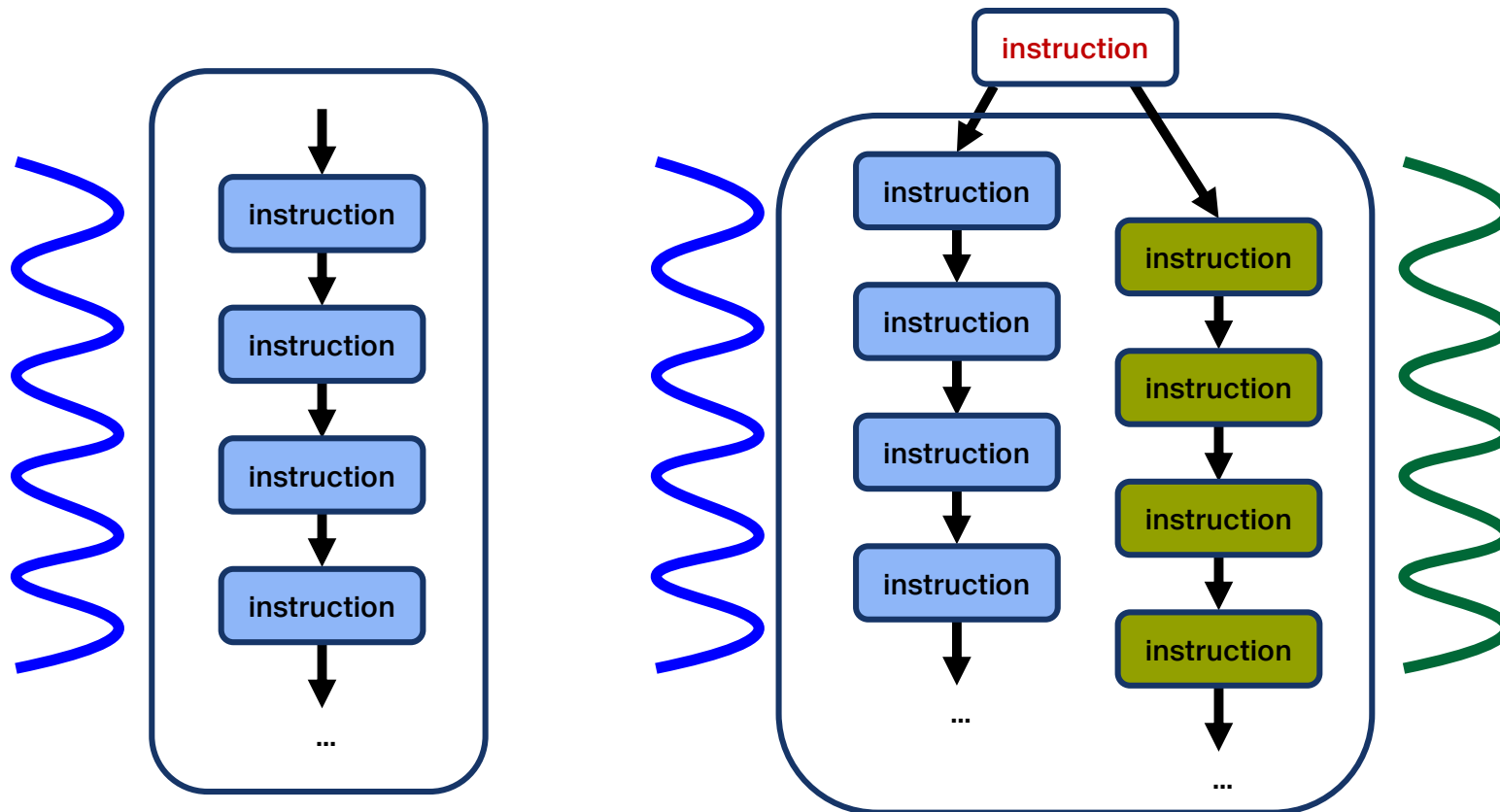
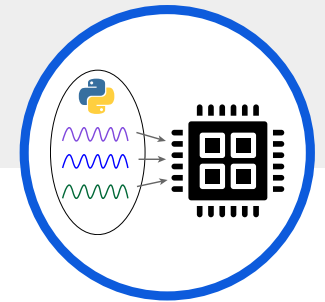


- Introduction à la programmation fonctionnelle :
  - Compréhensions de listes
  - Fonctions d'ordre supérieur
  - Lambdas

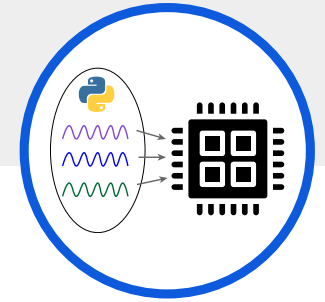
# Introduction à la programmation multitâche



# Introduction à la programmation multitâche



# Les threads en Python



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

def say_hello_periodically() -> None:
    thread_name = current_thread().name
    for i in range(5):
        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 éléments nécessaires

■ Cette fonction sera indiquée aux nouveaux threads pour exécution

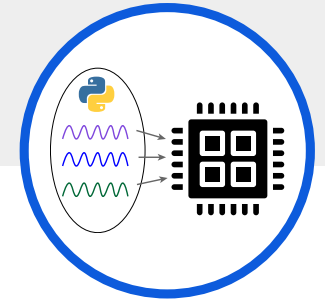
■ On dit bonjour en affichant le nom du thread

■ On attend une seconde avant de répéter la boucle

■ Deux nouveaux threads dont la tâche est d'exécuter le code préparé ci-dessus.

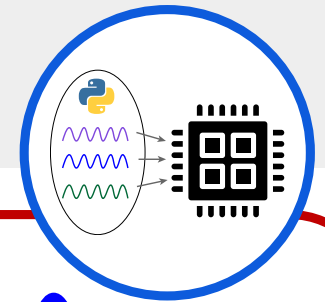
■ On démarre enfin les threads. L'exécution de la ligne suivante commence immédiatement **et ne bloque pas** : le reste du programme continue à s'exécuter sans attendre.

# Les threads en Python

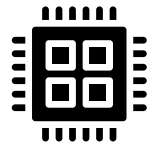


- Un **Thread** est un « fil d'exécution »
- Construit avec une **fonction à exécuter** pour lui dire quoi faire
  - Plusieurs threads peuvent utiliser la même fonction, comme dans l'exemple d'avant
- On **n'appelle pas** la fonction avec ( ), mais on indique juste son nom
- On le démarre en appelant 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

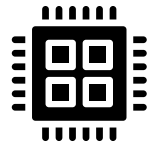
# Introduction à la programmation multitâche



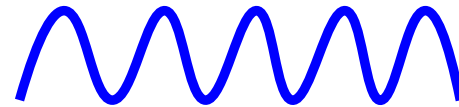
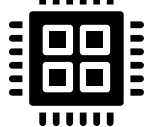
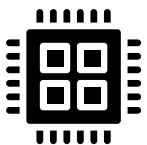
- Exécution séquentielle



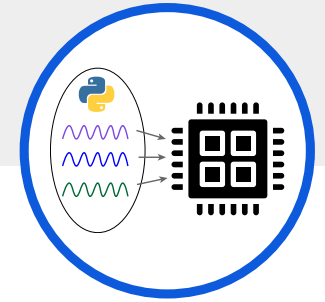
- Exécution concurrente



- Exécution parallèle

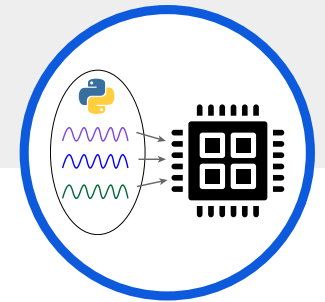


# Introduction à la programmation multitâche



- Le code des threads s'exécute en **parallèle** – ou plus précisément, de manière **concurrente**
  - Il y a parallélisme réel seulement si **plusieurs processeurs** (ou cores) sont disponibles
  - Sinon, un petit bout de chaque thread est exécuté à tour de rôle (**round-robin**)
- La planification des threads est gérée par le **système d'exploitation**, en tenant compte des autres programmes en cours
  - Il est **impossible de prédire l'ordre exact** d'exécution entre plusieurs threads
  - Cela pose des problèmes lorsque plusieurs threads accèdent à des **données partagées**

# Le problème des données partagées



```
class BankAccount:
    def __init__(self) -> None:
        self.balance: int = 1000

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

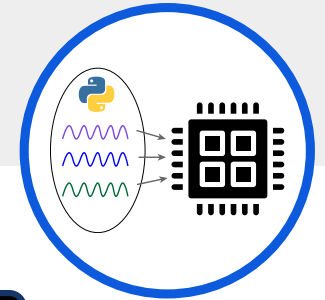
- Vous déposez 200 fr. sur un thread du système de transactions bancaires :  
`account.change_balance(delta=200)`

- Votre partenaire retire 100 fr. en même temps « dans un autre thread » :  
`account.change_balance(delta=-100)`

## Quel est le solde final ?

- En théorie, on s'attend à :  $1000 + 200 - 100 = 1100$ .
- Mais en pratique, à cause de l'exécution concurrente **sans protection**, le solde final peut être incorrect.
- Cela dépend de l'**interleaving** ! Le résultat peut être 1000, 1100 ou même 900.
- C'est une condition de course (**race condition**).

# Le problème des données partagées



## ■ Scénario 1

**balance : 1000**

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

**balance : 1200**

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

**balance : 1100**

## ■ Scénario 2

**balance : 1000**

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

**balance : 1000**

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

**balance : 900**

```
self.balance = new_balance
```

**balance : 1200**

## ■ Scénario 3

**balance : 1000**

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

**balance : 1000**

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

**balance : 1000**

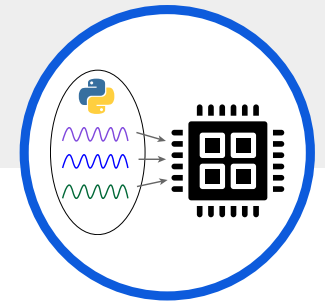
```
self.balance = new_balance
```

**balance : 1200**

```
self.balance = new_balance
```

**balance : 900**

# Le problème des données partagées : locks



```
from threading import Lock
```

```
class BankAccount:
```

```
    def __init__(self) -> None:
```

```
        self.balance: int = 1000
```

```
        self.lock = Lock()
```

```
    def change_balance(self, delta: int) -> None:
```

```
        with self.lock:
```

```
            new_balance = self.balance + delta
```

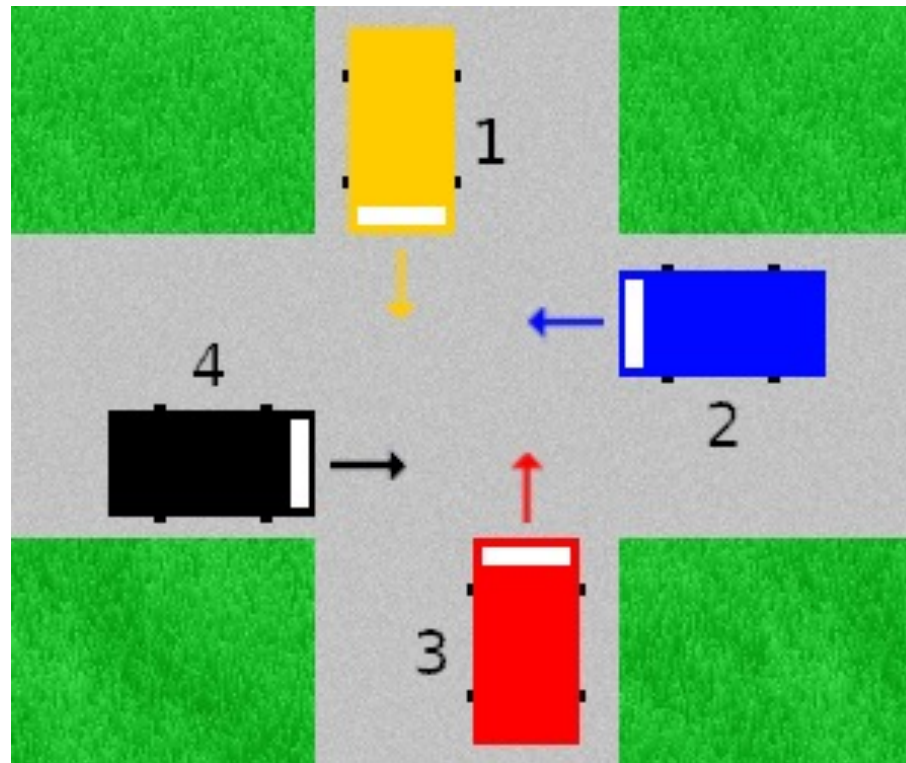
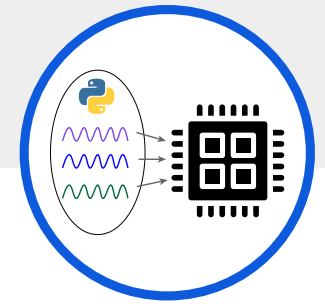
```
            self.balance = new_balance
```

■ Un **verrou** est utilisé pour protéger une section de code : on « ferme à clé » pendant qu'un thread travaille sur une ressource partagée.

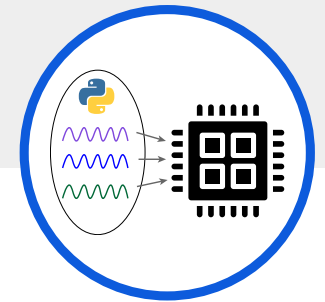
■ Le code protégé ne s'exécute que lorsque le verrou est disponible : on n'entre que si la « porte est déverrouillée ».

- Désormais, un seul thread peut exécuter la méthode à la fois : le premier thread à acquérir le verrou exécute la méthode **jusqu'au bout**, les autres **attendent** que le verrou soit libéré.
- Cela fonctionne dans ce cas simple, mais dans du code plus complexe, plusieurs threads peuvent **s'attendre mutuellement**. Si aucun ne progresse, on parle alors de **blocage mutuel (deadlock)**.

# Deadlock dans la vie de tous les jours



# Exemple : Multithreading dans une application graphique



- Le thread principal est utilisé par l'interface graphique pour attendre et réagir aux **événements** (clavier, souris, etc.)
- Pour toute opération de traitement longue, on crée un **thread séparé**
- Cela permet à l'interface graphique de **rester réactive**
- Par exemple : un bouton « **Stop** » peut interrompre une opération en cours **sans bloquer l'interface**

# Résumé Semaine 12 – ICC-P

- 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

[rafael.pires@epfl.ch](mailto:rafael.pires@epfl.ch)



**EPFL**

**Merci**