

## Information, Calcul et Communication

### CS-119(k) ICC – Programmation Semaine 11

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

# Précédemment, dans... ICC-P

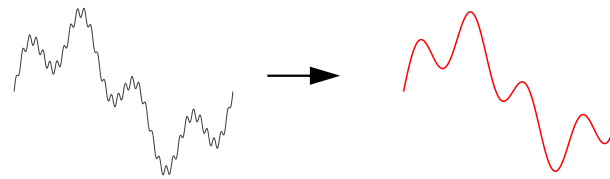
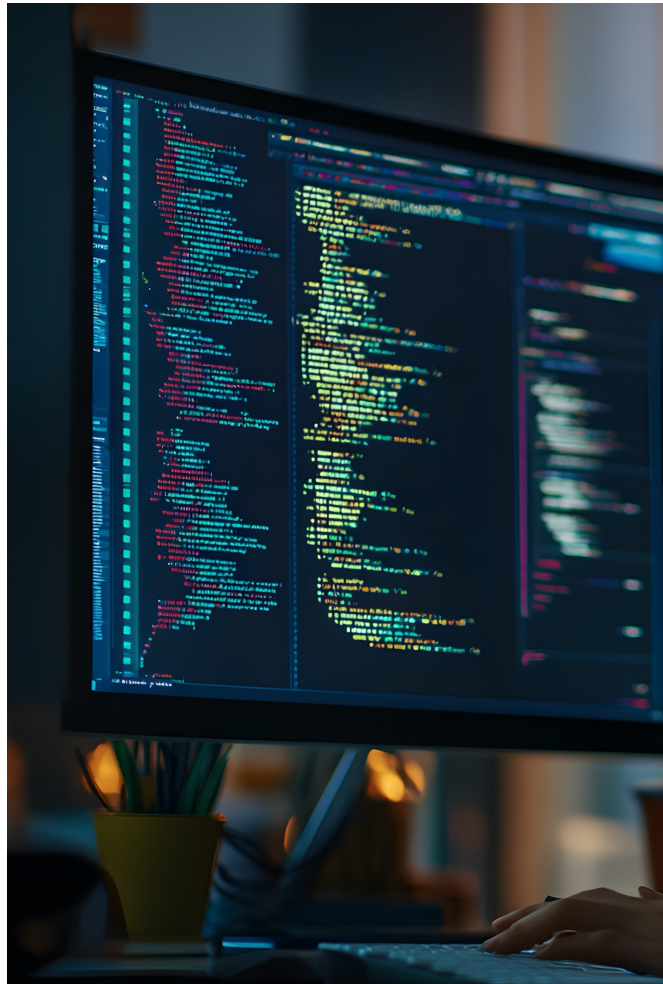


image de départ

50	40	60	100	90
20	30	80	90	70
40	20	40	30	50
30	10	50	20	30
20	20	10	30	40

kernel 3×3

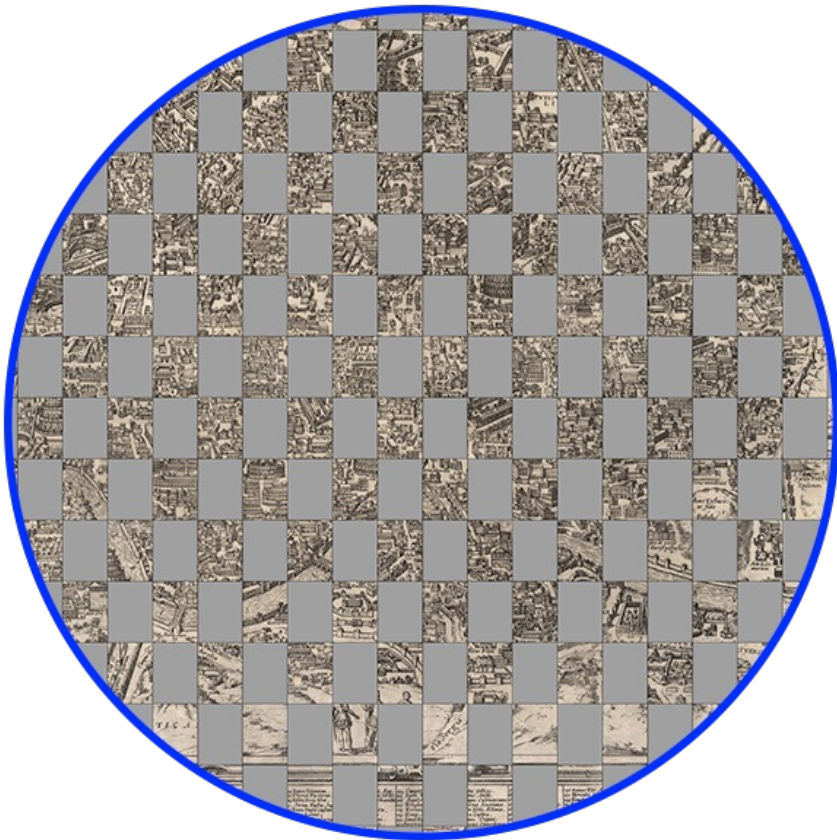
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{2}{10}$	$\frac{1}{10}$
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$

image après convolution

49	56	7
34	41	41
25	28	32

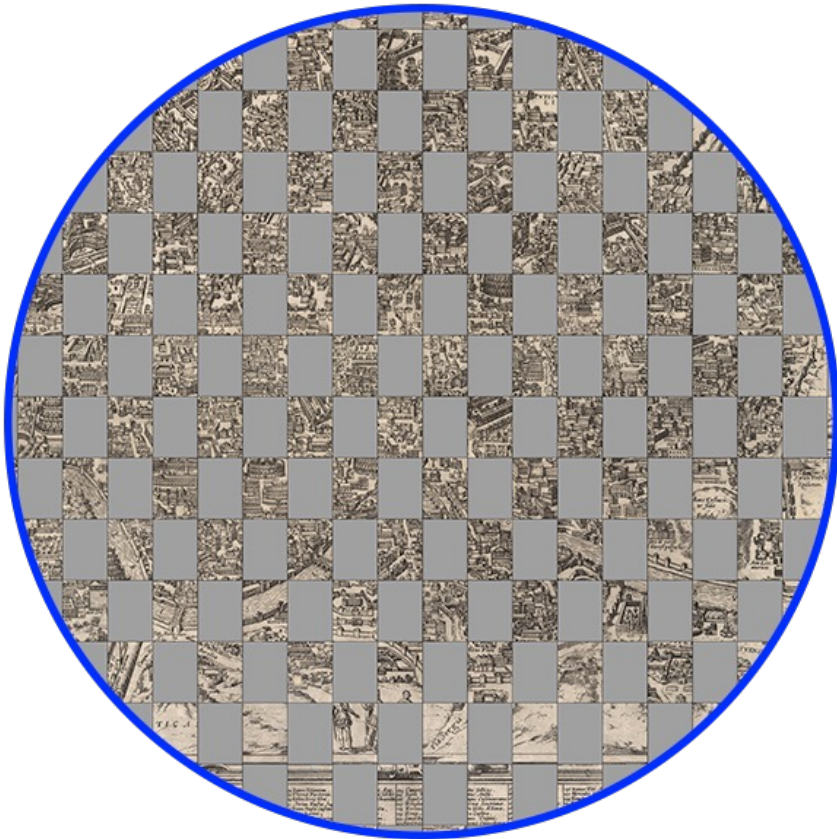
$$30 \frac{1}{10} + 80 \frac{1}{10} + 90 \frac{1}{10} + 20 \frac{1}{10} + 40 \frac{2}{10} + 30 \frac{1}{10} + 10 \frac{1}{10} + 50 \frac{1}{10} + 20 \frac{1}{10} = 41$$

# Mini-Projet



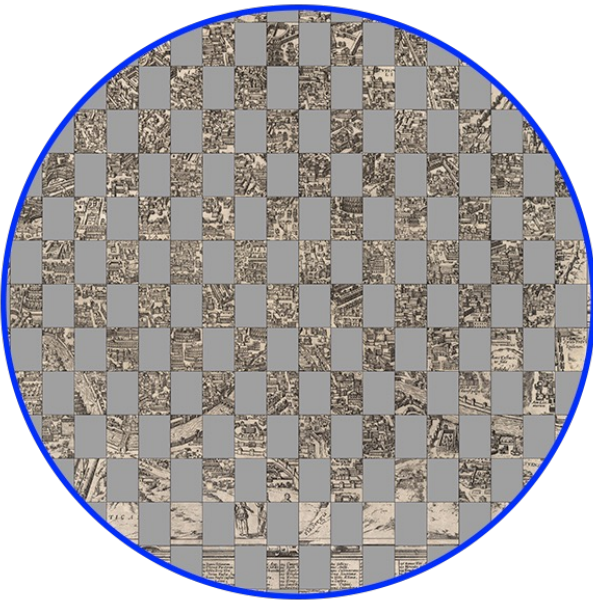
- **Étape A : manipulation d'images et puzzle**
  - Partie 1 : conversion RGB en niveaux de gris et flou
  - Partie 2 : reconstruction d'un puzzle en damier par comparaison des bords entre tuiles voisines
- **Étape B : localisation d'un motif**
  - Partie 1 : force brute
  - Partie 2 : recherche accélérée

# Mini-Projet : Étape B



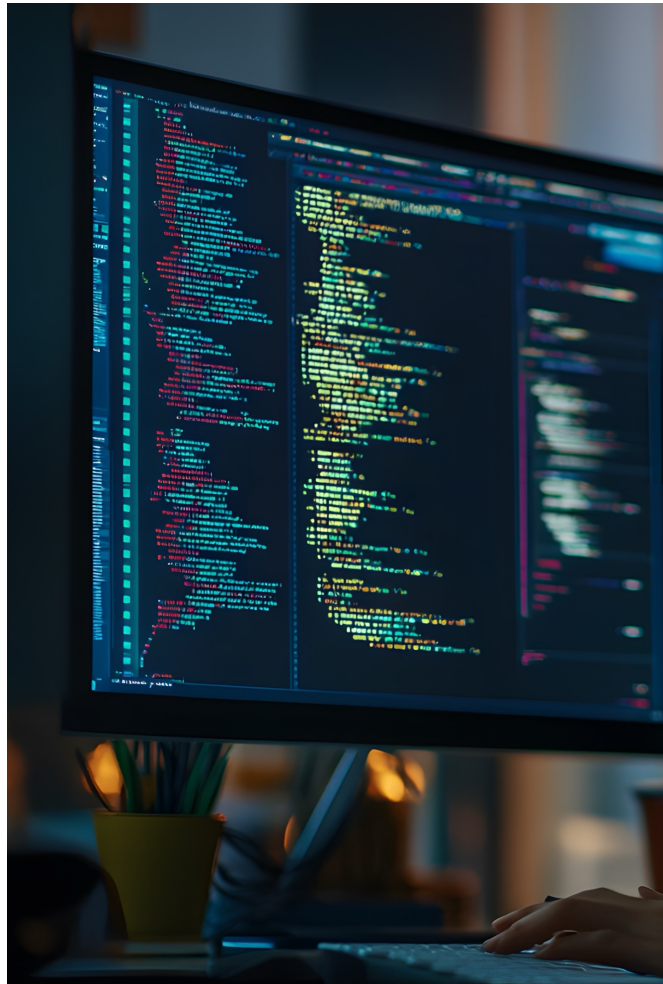
- **Partie 1 : recherche par force brute**
  - Score de similarité 2D entre le motif et chaque région de la carte
  - Glissement du motif sur toutes les positions valides, position de score minimal retenue
  - Surlignage du résultat par un bord rectangulaire
- **Partie 2 : recherche accélérée**
  - Réduction de l'image et du motif: flou de boîte puis sous-échantillonnage par un facteur
  - Recherche grossière sur les versions réduites, meilleures positions retenues
  - Affinage local autour de chaque candidat dans un voisinage à pleine résolution
  - Même position trouvée que la force brute, en une fraction du temps

# Mini-Projet : logistique



- Date limite de soumission du projet : **23 mai à 23h59**
  - soumettez tôt, vous pouvez resoumettre autant de fois que vous le souhaitez !
- Deux fichiers à soumettre (**un seul** rendu par groupe) :
  - `miniproject_a.py`
  - `miniproject_b.py`
- **Règles importantes :**
  - Modifiez **uniquement** `miniproject_a.py` et `miniproject_b.py` – ne modifiez pas les signatures de fonctions, `miniprojectutils.py`, `test_miniproject_a.py` ou `test_miniproject_b.py`.
  - Pour une soumission réussie, vous devez **compléter tous les commentaires TODO** dans `miniproject_a.py` et `miniproject_b.py`. Chaque TODO indique une section de code que vous devez implémenter – n'en laissez aucun inachevé.
  - Toute fonction auxiliaire que vous ajoutez doit être présente dans `miniproject_a.py` ou `miniproject_b.py`.
  - Évaluation : seules vos fonctions complétées seront évaluées, pas le bloc principal.
- Les inscriptions aux groupes sont closes – contactez votre assistant en cas de besoin.

# Précédemment, dans... ICC-P



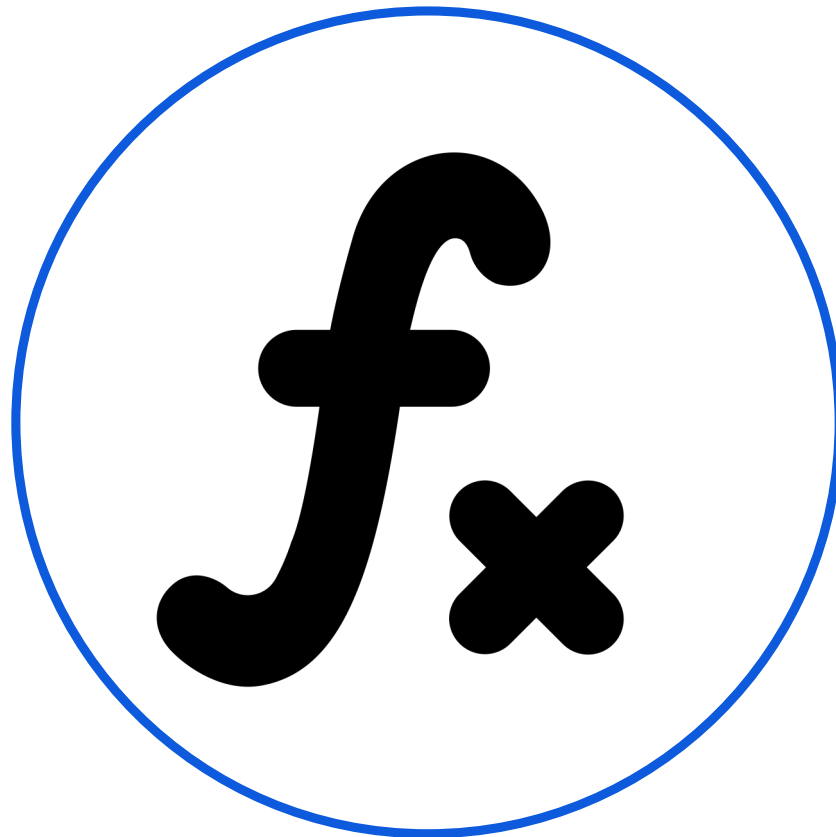
## Données

- Types de base en Python (immuables) : **int**, **float**, **str**, **bool**
- Types modifiables
  - **Listes**
  - **Sets**
  - **Dictionnaires**
  - **Dataclasses**
- **Fichiers** texte (encodage, modélisation, lecture, écriture)

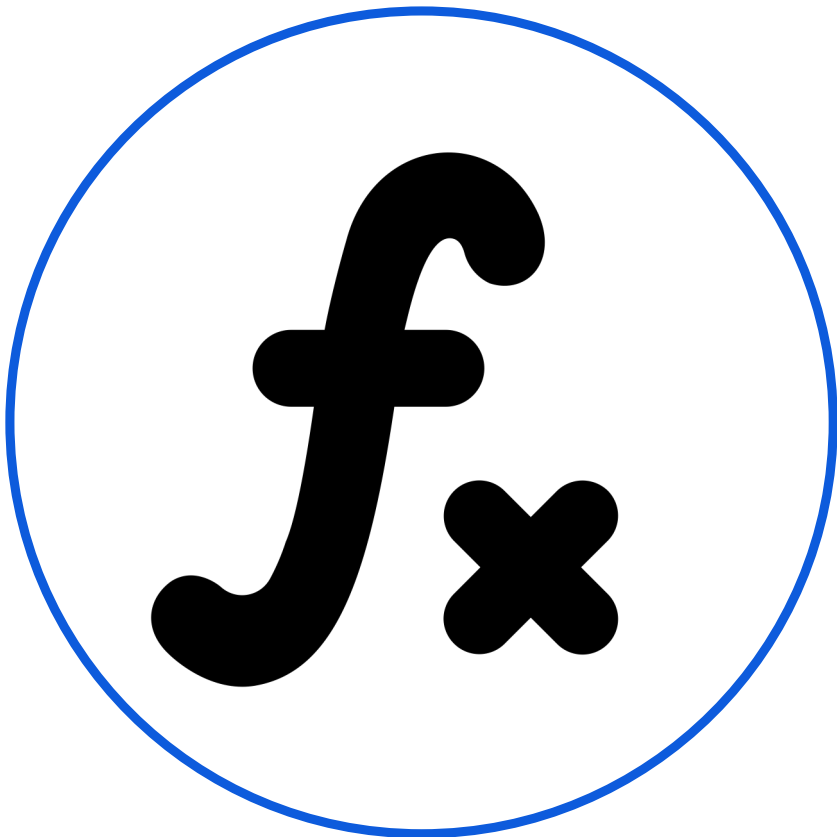
## Traitement

- **Méthodes**, **fonctions** et **slicing** pour calculer des valeurs dérivées
- **Branchements** pour exécuter du code selon la valeur d'une expression booléenne
- **Boucles** pour exécuter du code plusieurs fois
- **Fonctions**

# Introduction à la programmation fonctionnelle



# Introduction à la programmation fonctionnelle



- **Compréhensions de listes**
- Fonctions d'ordre supérieur
- Lambdas

# Compréhensions de listes



- Créez une liste de `int` qui indique la taille de chaque string issu d'une liste de strings donnée

```
words: list[str] = ["Elvis", "has", "left", "the", "building"]
size_of_words: list[int] = []
for word in words:
    size = len(word)
    size_of_words.append(size)

print(words)          # ['Elvis', 'has', 'left', 'the', 'building']
print(size_of_words) # [5, 3, 4, 3, 8]
```

- Compréhension de liste :

```
size_of_words = [len(word) for word in words]
```

# Compréhensions de listes



```
[expression for x in container if condition]
```

Expression qui sera évaluée pour créer chaque élément de la liste

Variable de boucle qui prendra toutes les valeurs des éléments contenus

▪ `range()`, `list`, `set`, ...

▪ Filtre (optionnel)

- Moyen concis de définition de listes (l'expression de gauche se lit **en dernier**)
- Plusieurs **for** enchaînables
- Possible de filtrer des itérations avec **if**
- Fonctionne aussi avec des sets et des dictionnaires

# Compréhensions de listes : exemples



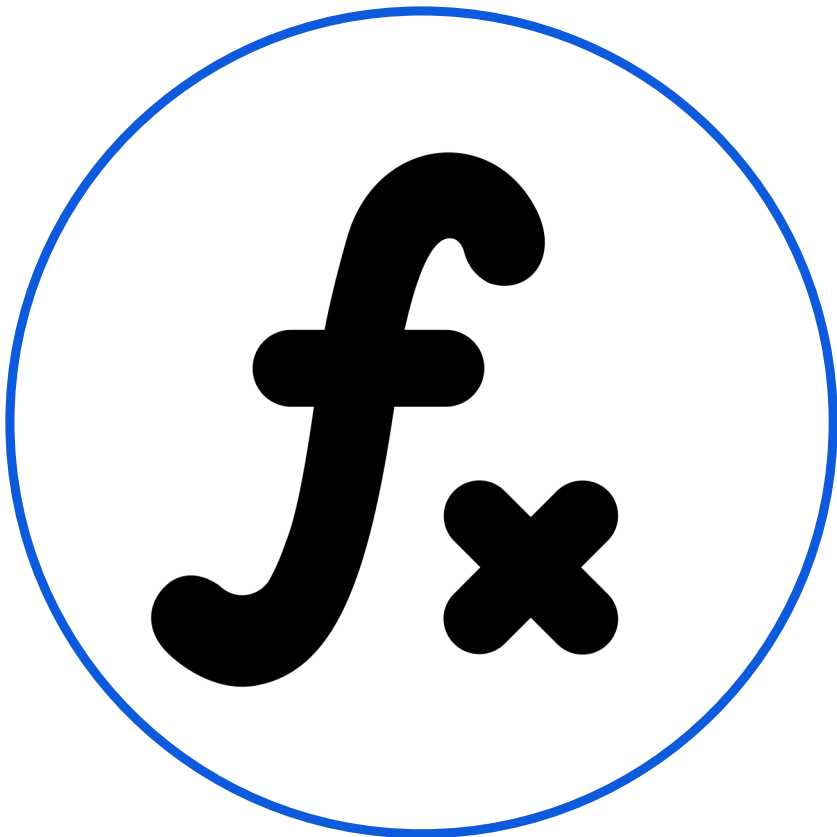
```
[0 for _ in range(10)] # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
my_ints = [2, 2, 5, 6, 1, 6, 3, 2]  
[x for x in my_ints if x > 3] # [5, 6, 6]
```

```
firsts = ["pomme", "banane"]  
seconds = ["banane", "orange", "kiwi", "pomme"]  
  
[f"{f}-{s}" for f in firsts for s in seconds if f != s]  
# ['pomme-banane', 'pomme-orange', 'pomme-kiwi',  
# 'banane-orange', 'banane-kiwi', 'banane-pomme']
```

```
[[x * y for y in range(1, 11)] for x in range(2, 11)]  
# [[2, 4, 6, ..., 18, 20], [3, 6, ..., 30], ..., [10, 20, ..., 100]]
```

# Introduction à la programmation fonctionnelle



- Compréhensions de listes
- **Fonctions d'ordre supérieur**
- Lambdas

# Fonctions : des valeurs comme les autres



- Nous avons utilisé des variables et paramètres pour stocker des valeurs « simples » (`int`, `float`, `list`, `set`, etc.)
- On peut aussi les utiliser pour stocker et manipuler directement des **fonctions**
- Le type `Callable` `[...], ...]`
  - Ce qui peut être appelé avec « (arguments) »
  - **Plusieurs types** pour les arguments, **un type** de retour

# Fonctions d'ordre supérieur



```
import math
from typing import Callable

def square(x: int) -> int:
    return x * x

print(square(4))

au_carré: Callable[[int], int] = square
print(au_carré(4))

def square2(x: int) -> int:
    return int(math.pow(x, 2))

def square3(x: int) -> int:
    return x ** 2

all_functions: list[Callable[[int], int]] = [square, square2, square3]
for f in all_functions:
    print(f(4))
```

# Fonctions d'ordre supérieur : paramètres



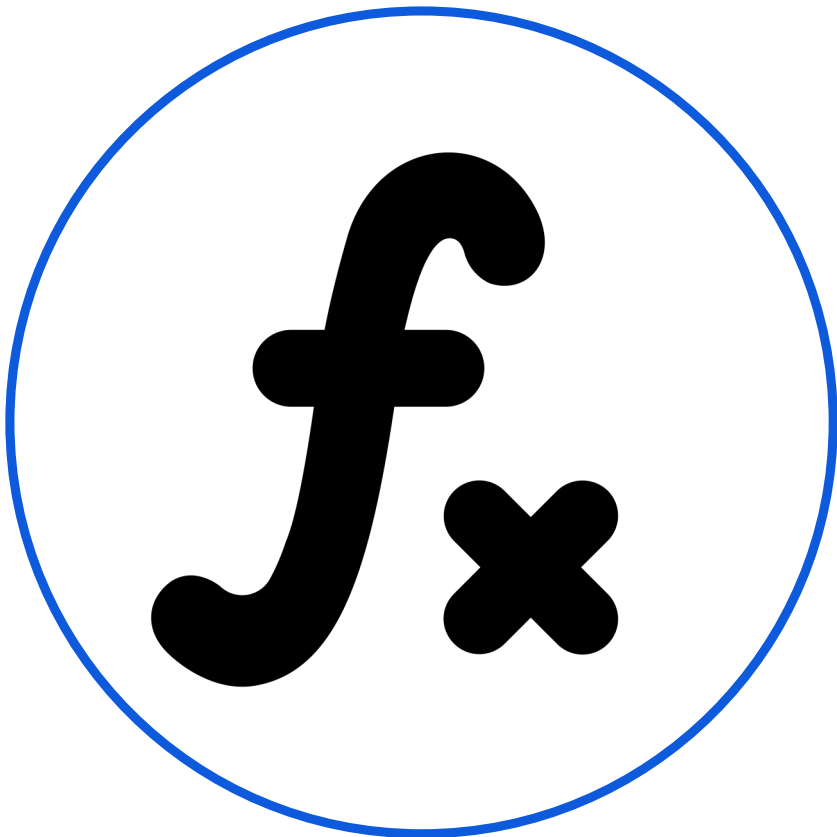
```
def apply_twice(f: Callable[[int], int], value: int) -> int:  
    return f(f(value))  
  
print(apply_twice(square, 4))
```

# Fonctions d'ordre supérieur : retour



```
def twice(f: Callable[[int], int]) -> Callable[[int], int]:  
    def f_of_f(x: int) -> int:  
        return f(f(x))  
    return f_of_f  
  
fourth_power = twice(square)  
print(fourth_power(4))
```

# Introduction à la programmation fonctionnelle



- Compréhensions de listes
- Fonctions d'ordre supérieur
- **Lambdas**

# Lambdas



```
def plus_2(x: int) -> int:  
    return x + 2  
  
plus_4 = twice(plus_2)
```

- Ici, on nomme la fonction **plus\_2** pour la passer à **twice** comme argument
- Peut-on faire plus concis ?

```
plus_4 = twice(lambda x: x + 2)
```

- **lambda** : fonction « anonyme »

# Lambdas

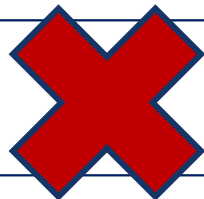


- Les fonctions lambda ...
  - N'ont pas de nom
  - Déclarent une liste d'arguments (sans type) avant les deux points
  - Ont à droite du deux-points une seule expression
  - Ont un return automatique

```
lambda x, y: x * y
```

```
lambda: 42
```

```
lambda x:  
    y = x + 2  
    return y * y
```



- Pas possible sur plusieurs lignes

# Lambdas



```
text = ... # un texte à analyser

def freq_analysis(text: str) -> dict[str, int]: ...
    ... # un dictionnaire qui relie chaque lettre à son nombre d'occurrences

frequencies: dict[str, int] = freq_analysis(text)
frequencies_as_tuples: list[tuple[str, int]] = list(frequencies.items())

sorted_frequencies: list[tuple[str, int]] = sorted(
    frequencies_as_tuples, key=lambda t: -t[1]
)

print(sorted_frequencies)
```

# Résumé Cours 11 – ICC-P

- Miniprojet : Étape B (et logistique)
- Introduction à la programmation fonctionnelle :
  - Compréhensions de listes
  - Fonctions d'ordre supérieur
  - Lambdas

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



**EPFL**

**Merci**