Information, Calcul et Communication

CS-119(k) ICC – Programmation
Semaine 6

Rafael Pires
rafael.pires@epfl.ch

Lausanne, 23.03.2026

Print, return et la récursivité

```python
def countdown_print(n):
    if n == 0:
        print("Terminé !")
    else:
        print(n)
        countdown_print(n - 1)
```

```python
countdown_print(3)
```

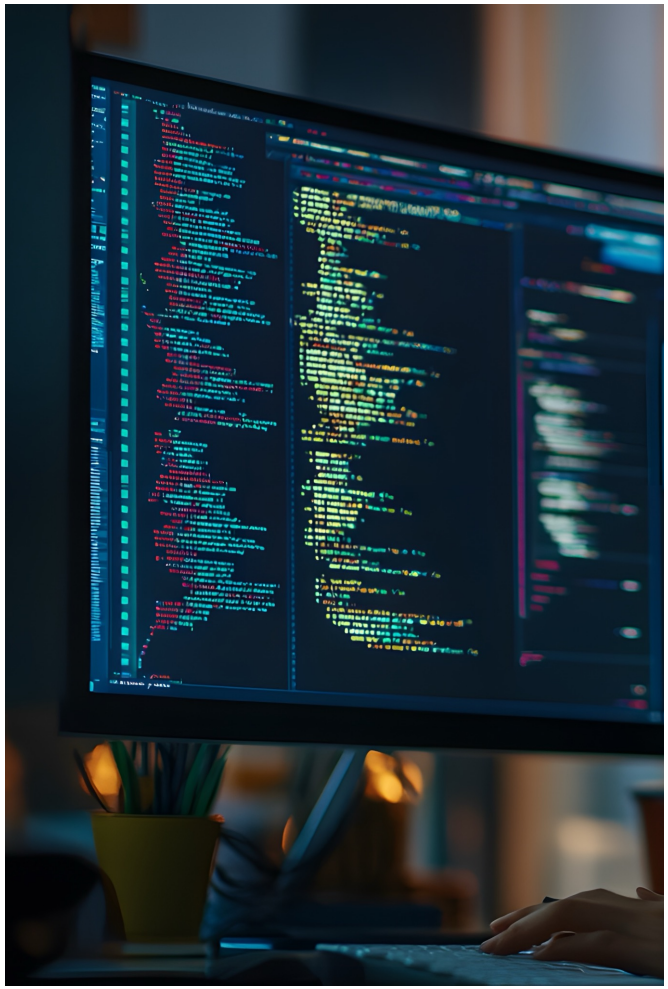
```python
# Résultat à l'écran :
# 3
# 2
# 1
# Terminé !
# La fonction ne renvoie pas de
# valeur exploitable.
```

```python
def countdown_return(n):
    if n == 0:
        return ["Terminé !"]
    else:
        return [n] + countdown_return(n - 1)
```

```python
resultat = countdown_return(3)
```

```python
print("La liste renvoyée est :", resultat)

# Ici, rien n'a été affiché dans la fonction elle-même.

# L'affichage vient de ce print externe :
# La liste renvoyée est : [3, 2, 1, 'Terminé !']
```

Précédemment, dans… ICC-P



Données

- Types de base en Python (immuables) : `int`, `float`, `str`, `bool`
- Types modifiables
  - Listes
  - Sets

```
values: list[int] = [1, 4, 2, 7, 3]
```

```
values: set[int] = {1, 3, 4, 2, 7, 2, 3}
```

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
  - Interruptions et saut dans les boucles avec `break` et `continue`
- Fonctions

Dictionnaire

Dictionnaire

- Un dict (dictionnaire) est une structure qui relie des clés à des valeurs.
- Conceptuellement, peut être vu comme un tableur à deux colonnes.
- Flexible sur la deuxième colonne :
  - Accepte objets modifiables, comme les listes et les sets

Clé	Valeur
"Alice"	15
"Bob"	17
"Charlie"	16

Dictionnaire

```python
ages: dict[str, int] = {"Alice": 15, "Bob": 17, "Charlie": 16}
print(ages) # {'Alice': 15, 'Bob': 17, 'Charlie': 16}

print(ages["Alice"]) # 15
# print(ages["Eve"]) # erreur d'exécution

if "Eve" in ages: # seulement si la clé est présente
    print(ages["Eve"])
else:
    print("n/a")
```

```python
ages["Dave"] = 15

print(ages)
# {'Alice': 15, 'Bob': 17, 'Charlie': 16, 'Dave': 15}

del ages["Dave"]
```

Clé	Valeur
"Alice"	15
"Bob"	17
"Charlie"	16
"Dave"	15

Exemple : Fréquence de mots

```
poem: list[str] = ... # une liste de mots

counts: dict[str, int] = {}

for word in poem:
    if word not in counts:
        counts[word] = 1
    else:
        counts[word] = counts[word] + 1

print(counts["les"])

for key in counts:
    print(f"{key} -> {counts[key]}")
```

Au début, le dictionnaire est vide

Pour chaque mot du poème, on regarde si le dictionnaire le connaît ou pas et on compte en fonction de ça

Les nouveaux mots ont un compteur de 1

Les nouveaux mots connus ont un compteur incrémenté

À la fin, le dictionnaire contient une valeur pour chaque mot différent du poème. On peut récupérer la valeur liée à un mot donné ainsi

On peut aussi itérer en demandant au dictionnaire de nous donner tous les mots qu'il connaît

Itération sur les dictionnaires

Clé	Valeur
"Alice"	15
"Bob"	17
"Charlie"	16

```
counts: dict[str, int] = {}
```

À des clés de type `str`, ce dictionnaire faire correspondre des valeurs de type `int`

```
for key in counts:
    value = counts[key]
    ...
```

Un `for-in` normal pour un dictionnaire donne la clé à chaque itération de la boucle

On peut bien sûr toujours récupérer la valeur

Clé	Valeur
"Alice"	15
"Bob"	17
"Charlie"	16

```
for key, value in counts.items():
    ...
```

Cette itération sur le résultat de la méthode `items()` donne à la fois la clé et la valeur à chaque itération!

Clé	Valeur
"Alice"	15
"Bob"	17
"Charlie"	16

```
for value in counts.values():
    ...
```

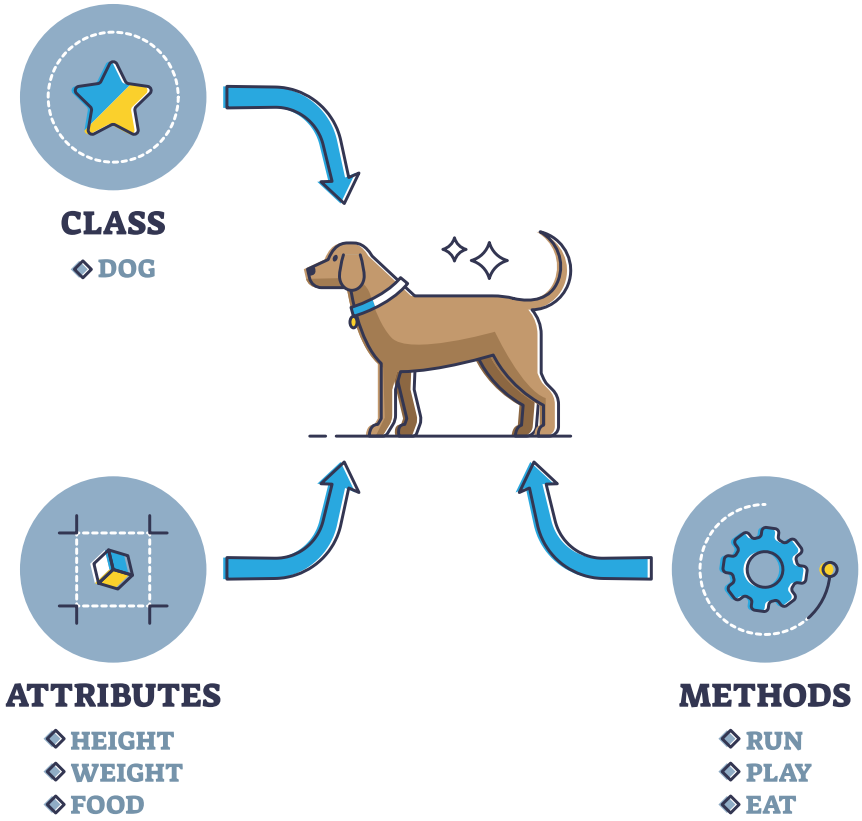
On peut aussi itérer seulement sur les valeurs, mais c'est moins fréquent, et on ne peut pas facilement récupérer la clé

Exemple : MyLittleFacebook

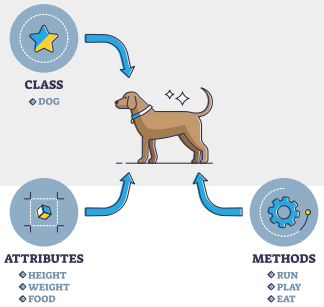
```python
# MyLittleFacebook
friendships: dict[str, set[str]] = {}

def add_friends(name1: str, name2: str) -> None:
    if name1 in friendships:
        friendships[name1].add(name2)
    else:
        friendships[name1] = {name2}
    if name2 in friendships:
        friendships[name2].add(name1)
    else:
        friendships[name2] = {name1}

add_friends("Alice", "Bob")
add_friends("Alice", "Charlie")
add_friends("Alice", "Dave")
add_friends("Charlie", "Eve")
```

Clé	Valeur
"Alice"	{"Bob", "Charlie", "Dave"}
"Bob"	{"Alice"}
"Charlie"	{"Alice", "Eve"}
"Dave"	{"Alice"}
"Eve"	{"Charlie"}

Dataclass

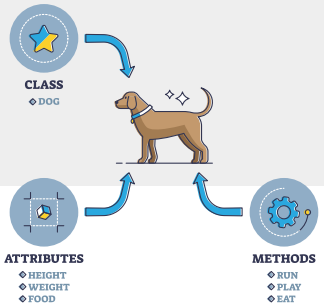

CLASS
DOG
ATTRIBUTES
HEIGHT
WEIGHT
FOOD
METHODS
RUN
PLAY
EAT

Dataclass



- Motivation : « *Je souhaite modéliser des types plus complexes, comme un objet* `Cylinder`*, en regroupant les données et les opérations associées en un seul endroit* ».
- Une classe modélise un objet de la vie réelle ou un concept abstrait.
- Une classe est un modèle pour un objet, en définissant ses données (les variables qui l'accompagnent) et ses opérations et calculs (ses méthodes)

Étape 0 : Sans classe, avec fonctions




```python
import math

def calc_cylinder_volume(radius: float, height: float) -> float:
    return math.pi * radius * radius * height

def calc_cylinder_surface_area(height: float, radius: float) -> float:
    a1 = 2 * math.pi * radius * height
    a2 = 2 * math.pi * radius * radius
    return a1 + a2

r = 1.2
h = 3.5
v = calc_cylinder_volume(r, h)
a = calc_cylinder_surface_area(r, h)
print(f"v = {v}, a = {a}")
```

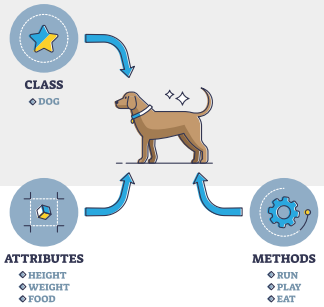
Les fonctions, avec arguments demandés et type de retour comme on les connaît déjà

Les valeurs qui définissent le cylindre

Les appels de fonctions qui font les calculs, en leur passant les arguments

Tout semble OK, mais il y a une erreur. Où est-elle?

Étape 1 : Avec classe et fonctions




```python
from dataclasses import dataclass

@dataclass
class Cylinder:
    radius: float
    height: float
```

« Je suis une nouvelle classe, **Cylinder** »

Chaque cylindre stocke deux floats comme « sous-variables ». Le premier s'appelle `radius`, le deuxième `height`

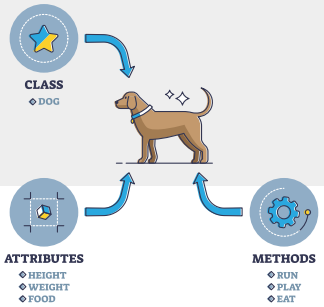
```python
def calc_cylinder_volume(cyl: Cylinder) -> float:
    return math.pi * cyl.radius * cyl.radius * cyl.height

def calc_cylinder_surface_area(cyl: Cylinder) -> float:
    a1 = 2 * math.pi * cyl.radius * cyl.height
    a2 = 2 * math.pi * cyl.radius * cyl.radius
    return a1 + a2

cyl = Cylinder(1.2, 3.5)
v = calc_cylinder_volume(cyl)
a = calc_cylinder_surface_area(cyl)
print(f"v = {v}, a = {a}")
```

Les fonctions ne demandent plus qu'un argument de type Cylinder. Plus de risque de confusion entre `r` et `h` !

Les fonctions utilisent la notation `objet.champ` pour récupérer les valeurs stockées par un `Cylinder`. Un `champ` est une « sous-variable »

Une variable d'un nouveau type, `Cylinder`, défini par nous ! Il stocke le rayon et la hauteur ensemble

Étape 2 : Avec classe et méthodes




```python
@dataclass
class Cylinder:
    radius: float
    height: float

    def calc_volume(self) -> float:
        return math.pi * self.radius * self.radius * self.height

    def calc_surface_area(self) -> float:
        a1 = 2 * math.pi * self.radius * self.height
        a2 = 2 * math.pi * self.radius * self.radius
        return a1 + a2


cyl = Cylinder(1.2, 3.5)
v = cyl.calc_volume()
a = cyl.calc_surface_area()
print(f"v = {v}, a = {a}")
```

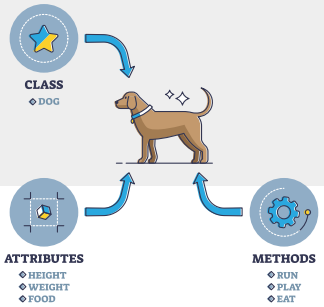
Les méthodes ont accès aux champs (sous-variables) stockées par l'objet (`radius` et `height`) pour faire leurs calculs via la référence à `self`, l'objet sur lequel elles ont été appelées. Ici, elles n'ont donc pas besoin de paramètres supplémentaires!

Les méthodes sont indentées car déclarées dans la classe

Plus de fonctions « en vrac » dans le code; on appelle les méthodes que la classe `Cylinder` définit

Le paramètre **`self`** est fourni automatiquement pour l'appel de méthodes. Ici, il vaut **`cyl`**

Appels avec noms des paramètres




```python
from dataclasses import dataclass

@dataclass
class Cylinder:
    radius: float
    height: float


cyl = Cylinder(1.2, 3.5)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(radius=1.2, height=3.5)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(height=3.5, radius=1.2)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(3.5, 1.2)
print(cyl) # Cylinder(radius=3.5, height=1.2)
```

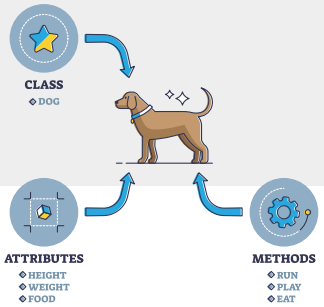
Arguments positionnels : la position compte

Arguments nommés : variante plus lisible conseillée

Peuvent être réordonnés !

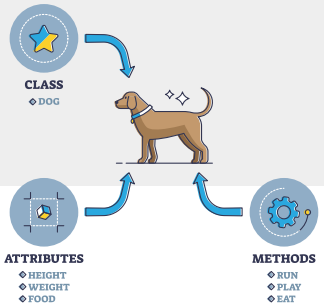
Réordonner des arguments positionnels change la sémantique du code

Dataclass : Concepts principaux



- modélise un objet (ou un concept abstrait) précis
- a un nom
- stocke des données avec ses champs (sous-variables)
- fournit des méthodes pratiques
  - Les méthodes ont toujours comme premier paramètre `self`, une référence à l'objet sur lequel la méthode est appelée
  - Les méthodes ont accès aux champs via le paramètre `self`

Dataclass : Self



- Le premier paramètre de chaque méthode est `self`
- Il n'est pas spécifié lors de l'appel, mais fourni automatiquement
  - Il référence toujours l'objet sur lequel la méthode est appelée
- Il sert à lire ou écrire des champs ou à appeler d'autres méthodes du même objet
- Par convention, on ne spécifie pas son type
- On peut toujours ajouter d'autres paramètres après si nécessaire pour que la méthode fasse son travail — comme on l'a appris pour les fonctions
  - Des valeurs pour ces autres paramètres doivent être fournies lors de l'appel

Résumé Cours 6 – ICC-P

- Un dictionnaire (de type `dict[K, V]`) fait correspondre des clés à des valeurs
  - Avec un `for-in` normal, on itère sur les clés d'un dictionnaire
  - La recherche de la valeur correspondant à une clé donnée est optimisée
- Une classe définit un nouveau type
- Une classe déclare ses champs (sous-variables) et peut définir des méthodes en plus
- Les méthodes sont liées à la classe et s'appellent « sur » une variable du type de la classe
- Pour chaque méthode, le paramètre automatique `self` est toujours à mentionner en premier lors de la définition de la méthode
  - Il est fourni automatiquement lors de l'appel
- Les méthodes ont accès aux champs (avec la notation `self.xyz`) et peuvent aussi changer leurs valeurs

rafael.pires@epfl.ch

Merci