EPFL

# ICC

# Programmation

SIE/CGC, **Cours 6**

25 mars 2024

*Jean-Philippe Pellet*

# *Previously, on 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: `if <condition>: ... else: ...` et ses variantes
- Boucles pour exécuter du code plusieurs fois:
  - Boucle `while <condition>: ...`
  - Boucle `for i in range(...): ...`
- Déclaration de fonctions avec type de retour et paramètres:
  - `def calculate_area(r: float) -> float: return ...`
- Utilisation de listes
- Utilisation de sets
  - Ce sont des objects modifiables

# Dans deux semaines: Midterm

- Le 8 avril 2024 à 15h en salles CO020, CO021, CO023 (pas CE14)
  - Connectez-vous dès 15h00, on commence à 15h15.
  - Fin: 18h00 (165 minutes — *pas d'aménagement nécessaire*)
- Sur les machines virtuelles
  - **Testez vos accès aux machines virtuelles**! En cas de souci: 1234@epfl.ch
  - **Setup de VS Code sur les VM** à faire selon instructions de la semaine 1
- Partie QCM sur Moodle, partie code sur VS Code
- Vous avez droit à une page A4 (recto, pas de verso) de résumé personnel
  - Recommandé: manuscrit. Possiblement fait à l'ordi, mais personnel. Pas de screenshots de slides ou autre

# Cours de cette semaine

*Dictionnaires*

*Dataclasses*

# Dictionnaire

- Un dict (dictionnaire) est une structure qui relie des clés à des valeurs
- Conceptuellement, peut être vu comme un tableur à deux colonnes
  - En plus flexible surtout sur la deuxième colonne
- Objets modifiables (non immuables) comme les listes et les sets

| Clé | Valeur |
|---|---|
| `"Alex"` | `10` |
| `"Émelyne"` | `9` |
| `"Victor"` | `6` |

*Démo*

# Dictionnaire: exemple

```python
ages: dict[str, int] = {"Alex": 8, "Émelyne": 7, "Victor": 4}
print(ages)           # {'Alex': 8, 'Émelyne': 7, 'Victor': 4}

print(ages["Alex"])       # 8
# print(ages["Sandra"])   # erreur d'exécution

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

ages["Nathan"] = 4

print(ages)
# {'Alex': 8, 'Émelyne': 7, 'Victor': 4, 'Nathan': 4}

del ages["Nathan"]
```

| Clé | Valeur |
|---|---|
| "Alex" | 10 |
| "Émelyne" | 9 |
| "Victor" | 6 |
| "Nathan" | 5 |

# Autre exemple: fréquence de mots

```python
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

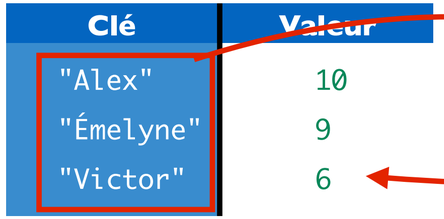
À la fin, le dictionnaire contient une valeur pour chaque mot différent du poème. On peut récupérer la valeur liée à un mot donné ainsi

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

# Itération sur les dictionnaires

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

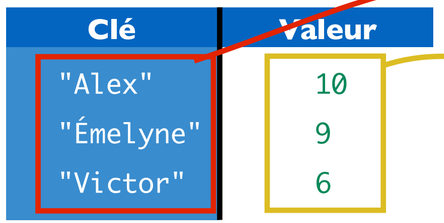
À des clés de type *str*, ce dictionnaire faire correspondre des valeurs de type *int*




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

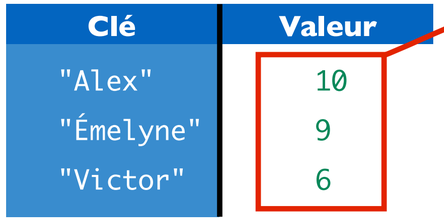
Un *for-in* normal pour un dictionnaire donne la clé à chaque itération de la boucle

On peut bien sûr toujours récupérer la valeur




```
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!




```
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 plus intéressant: *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}
```

```python
add_friends("Alex", "Victor")
add_friends("Alex", "Emelyne")
add_friends("Alex", "Emelyne")
add_friends("Emelyne", "Rose")
```

| Clé | | Valeur associée |
|---|---|---|
| `"Alex"` | → | `{"Victor", "Emelyne"}` |
| `"Victor"` | → | `{"Alex"}` |
| `"Emelyne"` | → | `{"Rose", "Alex"}` |
| `"Rose"` | → | `{"Emelyne"}` |

# Cours de cette semaine

*Dictionnaires*

*Dataclasses*

# Motivation

- Classes:
  *«Je veux modéliser des types plus complexes (par exemple un objet Cylinder) et rassembler les données et opérations y relatives en un seul endroit»*

# Classe = type plus avancé

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)

*Démo*

*Exemple suivi:*
*calcul du volume et de l'aire d'un cylindre*

# 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 + fonctions

```
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

«Je suis une nouvelle classe, *Cylinder*»

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

```
from dataclasses import dataclass

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

# Construction 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 réordrés!

Réordrer des arguments positionnels change la sémantique du code

# Étape 2: avec classe + méthodes

```
@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
```

Ces 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, elle n'ont donc pas besoin de paramètres supplémentaires!

Les méthodes sont indentées car déclarées *dans* la classe

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

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*

# Concepts principaux d'une classe

Une classe typique…

- 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

# 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

- 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), 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