# ICC: Programmation

GC/MX, Cours 6, 25 octobre 2019

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:
- Déclaration de fonctions avec type de retour et paramètres
- Utilisation de listes:
  - `values: List[int] = [1, 2, 2, 7, 3]`
- Utilisation de sets:
  - `unique_values: Set[int] = {1, 2, 2, 7, 3}`
- Utilisation de dictionnaires:
  - `ages: Dict[str, int] = {"A": 1, "E": 4, “V": 1}`

# Répétition: *MyLittleFacebook*

```
from typing import *

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

str: type pour les clés

Set[str]: type pour les valeurs

*«Je relies des strings à des ensembles de strings. Si tu me donnes un string, je peux rapidement te donner l'ensemble de strings correspondant»*

| Clé: str | | Valeur associée: Set[str] |
|---|---|---|
| "Alex" | → | {"Victor"}<br>"Emelyne"} |
| "Victor" | → | {"Alex"} |
| "Emelyne" | → | {"Rose"}<br>"Alex"} |
| "Rose" | → | {"Emelyne"} |

# Cours de cette semaine

*Tuples*

*Classes simples*

# Exemple: retourner plusieurs valeurs

*Démo*

*Avec l'exemple de divmod()*

# Exemple: retourner plusieurs valeurs

```
a, b = divmod(10, 3)
pair = divmod(10, 3)
a == pair[0], b == pair[1]

def mydivmod(q: int, r: int):
    return q // r, q % r
    return (q // r, q % r)

from typing import Tuple

def three_powers(n: int) -> Tuple[int, int, int]:
    return n, n * n, n * n * n

t = three_powers(4)
t0, t1, t2 = t
t2 == t[2]
```

divmod() semble retourner deux valeurs…

… mais retourne en fait une *paire de valeurs*, qu'on peut aussi directement assigner à une seule variable

Cette variable contient comme une liste fixe à deux cases. Ces 2 expressions sont vraies

On écrit des paires en séparant des valeurs par une virgule (on peut – des fois, doit – rajouter des parenthèses)

Généralisation: Tuple à *n* éléments.
Type: Tuple[type1, type2, …, type*n*]

Exemple de *tuple unpacking*: on assigne directement le tuple à 3 variables séparées en une fois…

… en fait, comme sur la première ligne

# Tuple ou liste?

- Si taille inconnue → liste
- Si taille connue, mais grande → liste
- Si éléments typiquement hétérogènes → tuple
- Si structure doit être modifiée → liste
- Si structure immuable → tuple
- Si structure doit être modifiée, hétérogène, de petite taille → classe

# Cours de cette semaine

*Tuples*

*Classes simples*

# Motivation

- Classes:
  *«Je veux modéliser un 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, méthodes statiques

```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 méthodes 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, méthodes statiques

```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*

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

«Je suis une nouvelle classe, *Cylinder*»

«On me construit en me donnant deux floats»

```python
class Cylinder:
    def __init__(self, radius: float, height: float):
        self.radius = radius
        self.height = height
```

«Je stocke deux doubles. Le premier s'appelle *radius*, le deuxième *height*»

# Étape 1b: classe avec __repr__

```
cyl = Cylinder(1.2, 3.5)
print(cyl)  # <__main__.Cylinder object at 0x10202f0f0>
```

Représentation peu informative…

```
class Cylinder:
    def __init__(self, radius: float, height: float):
        self.radius = radius
        self.height = height

    def __repr__(self) -> str:
        return f"Cylinder(radius={self.radius}, height={self.height})"
```

Méthode spéciale __repr__: «voici comment me représenter en tant que string» (par exemple sur le terminal pour *print*)

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

Mieux!

# Construction avec noms des paramètres

```python
class Cylinder:
    def __init__(self, radius: float, height: float):
        self.radius = radius
        self.height = height

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 de classe

```
class Cylinder:
    def __init__(self, radius: float, height: float):
        self.radius = radius
        self.height = height

    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}")
```

Ces méthodes ont accès aux variables d'instance 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!

Plus de fonction «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 variables d'instance
- fournit des méthodes (non-statiques) 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 variables d'instance via le paramètre self
- a une méthode spéciale __init__
  - — __init__ peut avoir des arguments, qui donnent en général la valeur initiale aux variables d'instance

# 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 variables d'instances ou à appeler d'autres méthodes du même objet
- Par convention, on ne spécifie pas son type

# Résumé Cours 6

- Des tuples servent à regrouper des valeurs qui vont ensemble
- Une classe définit un nouveau type
- Une classe déclare (habituellement) ses variables d'instance dans la méthode spéciale __init__, peut définir des méthodes en plus
- Pour chaque méthode, le paramètre automatique self est toujours en premier
- Les méthodes ont accès aux variables d'instance et peuvent aussi changer leurs valeurs