

ENG-209

Data science pour ingénieurs avec Python

Cours 2: Structures de données

Jean-Philippe Pellet

23 septembre 2024

Cours 2

Structures de données:
tuple, list, set, dict, dataclass

Cours 2

Structures de données:
tuple, list, set, dict, dataclass

Tuples — plusieurs valeurs ensembles

```
my_data = ("Jean-Philippe", 1.79, 3)
```

Un tuple peut être défini avec des parenthèses.

```
n = len(my_data) # 3
```

On peut demander le nombre d'éléments d'un tuple

```
name = my_data[0]  
nb_kids = my_data[2]
```

On récupère des éléments du tuple avec [x]

```
my_data[2] = 4
```

Les tuples sont immuables (éléments pas réaffectables)

```
two_primes = 2, 3
```

La plupart du temps, on peut aussi faire sans parenthèses

```
a, b = b, a
```

Voici comment «swapper» deux variables en Python avec des tuples

```
for item in my_data: # Jean-Philippe  
    print(item)      # 1.79  
                    # 3
```

On peut itérer à travers des tuples avec un for normal

Tuples — fonctions et type

```
def to_hours_and_minutes(hours_dec):  
    hours_int = int(hours_dec)  
    minutes = int((hours_dec - hours_int) * 60)  
    return hours_int, minutes
```

Les tuples sont pratiques pour retourner plusieurs valeurs d'une fonction

```
h_and_min = to_hours_and_minutes(2.5)
```

La fonction peut être appelée normalement et retourne un tuple, ici de longueur 2

```
h, m = to_hours_and_minutes(2.5)
```

Le tuple peut être directement «déstructuré»: ici, on assigne les deux variables d'un coup

```
print(f"{h} h {m} min")
```

Les tuples ont le type générique tuple[...] avec autant d'indication de type qu'il y a d'éléments dans le tuple

```
def to_hours_and_minutes2(hours_dec: float) -> tuple[int, int]:  
    ...
```

«Cette fonction retourne un tuple de deux int»

Cours 2

Structures de données:
tuple, **list**, set, dict, dataclass

Listes — bases

```
temperatures = [28.7, 25.3, 22.0, 23.9, 25.8, 30.1]
```

Une liste peut être définie avec des []

```
print(temperatures[0])
```

On accède à ses éléments avec [] également, comme pour les tuples, en passant un index

```
temperatures[0] = ...
```

Chaque «case» de la liste peut être modifiée

```
len(temperatures) # 6
```

On peut récupérer sa longueur

```
avg = 0.0
```

```
for i in range(len(temperatures)):
```

```
    avg += temperatures[i]
```

```
avg /= len(temperatures)
```

On peut itérer à travers, par exemple pour calculer sa moyenne...

```
avg = 0.0
```

```
for t in temperatures:
```

```
    avg += t
```

```
avg /= len(temperatures)
```

... mais à la place de range(len(...)), on utilisera plutôt une itération directe sur les éléments

Listes — fonctions et méthodes

```
temperatures = [28.7, 25.3, 22.0, 23.9, 25.8, 30.1]
```

```
max(temperatures) # 30.1
```

```
min(temperatures) # 22.0
```

Il y a une série de fonctions prédéfinies qui sont applicables aux listes: max, min, sum, etc.

```
from statistics import mean, stdev
```

```
mean(temperatures) # 25.966666666666667
```

```
stdev(temperatures) # 3.0011109054259673
```

Certaines autres fonctions (mean, stdev) font partie de la bibliothèque standard, mais sont à importer d'un module séparé

```
temperatures.append(28.2)
```

```
temperatures.extend((25.6, 22.8, 18.5))
```

```
temperatures.sort()
```

```
temperatures.clear()
```

Une série de méthodes existent pour modifier la liste et son contenu

Ajout d'un seul (append) ou de plusieurs éléments (extend, en passant ici un tuple) à la fin de la liste

Tri

Effaçage, etc.

Listes — subscripting, slicing

```
temperatures = [28.7, 25.3, 22.0, 23.9, 25.8, 30.1]
```

```
temperatures[0:3] # [28.7, 25.3, 22.0]
```

On peut extraire une partie des éléments en passant un slice à la place d'un index unique entre []

```
temperatures[:3] # idem
```

On peut omettre le premier index si c'est 0

```
temperatures[4:] # [25.8, 30.1]
```

On peut omettre le second index s'il faut aller jusqu'à la fin de la liste

```
temperatures[-2:] # idem
```

On peut utiliser des index négatifs pour faire référence à des éléments depuis la fin de la liste

```
temperatures[0:2] = [27.8, 23.5]
```

Avec des slices, on peut modifier plusieurs cases à la fois

```
temperatures[0:2] = []
```

On peut également en supprimer...

```
temperatures[0:0] = [27.8, 23.5]
```

... et en insérer (ici au début)

```
temperatures[0] = [27.8, 23.5]
```

Attention, ceci n'a pas le même effet, et stocke une liste complète à la position 0 de la liste principale

Listes — compréhensions de listes (1/2)

```
words = ["Elvis", "has", "left", "the", "building"]  
size_of_words = [len(word) for word in words]  
print(size_of_words) # [5, 3, 4, 3, 8]
```

Génération rapide d'une liste selon le format:

[<expr> for <generator>]

Très pratique pour créer des listes dérivées

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

On peut nommer la variable _ (simple convention) si

l'on n'a pas besoin de sa valeur

```
points = [2, 2, 5, 6, 1, 6, 3, 2]  
[p for p in points if p > 3]  
# donne [5, 6, 6]
```

*On peut sauter la génération de certaines valeurs avec
un filtre en incluant un if après le for*

```
range(0, 10, 0.1)  
[x / 10 for x in range(0, 100)]  
# [0.0, 0.1, 0.2, ..., 9.8, 9.9]
```

*Voici comment on pourrait générer une range de floats,
ce qui n'est pas possible avec la fonction de base*

Listes — compréhensions de listes (2/2) et type

```
firsts = ["Jean", "Pierre"]
seconds = ["Pierre", "Michel", "Marc", "Jean"]
[f"{f}-{s}" for f in firsts for s in seconds if f != s]
# ['Jean-Pierre', 'Jean-Michel', 'Jean-Marc',
#  'Pierre-Michel', 'Pierre-Marc', 'Pierre-Jean']
```

On peut avoir plusieurs générateurs et plusieurs filtres. Ici, on génère toutes les combinaisons de noms composés possibles tant que la première partie est différente de la seconde

```
[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]]
```

On peut générer des listes imbriquées, par exemple si l'expression déterminant la valeur de chaque case de la liste est elle-même une compréhension de listes

```
words: list[str] = ...
my_ints: list[int] = ...
```

Pour indiquer les types, on utilise list[...] avec un paramètre de type

```
mult_table: list[list[int]] = ...
```

Exemple d'un type de liste imbriquée

Cours 2

Structures de données:
tuple, list, **set**, dict, dataclass

Sets — bases

```
numbers = {2, 3, 3, 4, 2, 5}
print(numbers) # {2, 3, 4, 5}
```

```
print(numbers[0])
```

```
for n in numbers:
    print(n)
```

```
if 5 in numbers:
    print("5 is in the set")
    numbers.remove(5)
```

```
empty_set = set()
```

```
numbers: set[int] = ...
```

Un set peut être défini avec des { }

Chaque élément y apparaît au plus une fois

Impossible de récupérer l'élément n , pas d'ordre intrinsèque...

... mais possible d'itérer à travers

Opération optimisée ($O(1)$): déterminer si un élément est oui ou non dans un set donné

Une série de méthodes existent: remove(), add(), clear(), mais aussi difference(), intersection(), issuperset(), issubset()

Un set vide s'écrit set(), parce que {} est le dictionnaire vide, pas le set vide

Le type paramétrique est set[...]

Types linéaires, conversions

On peut convertir entre différentes structures de données, dans des cas simples avec list(...) et set(...)

```
list(range(10))  
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Conversion d'une range en liste

```
my_data = ("Jean-Philippe", 1.79, 3)  
list(my_data)  
# ['Jean-Philippe', 1.79, 3]
```

Conversion d'un tuple en liste hétérogène (pas forcément recommandé)

```
numbers = [2, 3, 3, 4, 2, 5]  
numbers = list(set(numbers))  
# [2, 3, 4, 5]
```

Conversion d'une liste en set puis de nouveau en liste afin d'avoir une liste normale mais sans valeurs répétées

Cours 2

Structures de données:
tuple, list, set, **dict**, dataclass

Dict — bases

```
ages = {"Alex": 11, "Emelyne": 9, "Victor": 7}
```

Un dict relie des clés (ici des noms) à des valeurs (ici des âges). Il peut être défini avec des { }, des : pour relier chaque clé à sa valeur

```
print(ages["Alex"])          # 11  
print(ages["Sandra"])
```

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

```
ages.get("Sandra", -1)
```

```
ages["Nathan"] = 5
```

```
del ages["Nathan"]
```

Il est optimisé pour retourner rapidement ($O(1)$) la valeur liée à une clé

À moins qu'on ne le sache pour sûr, on doit s'assurer de la présence d'une valeur avec l'opérateur in

On peut aussi utiliser la méthode get(), qui permet d'indiquer une valeur par défaut à retourner si la clé n'est pas trouvée

On peut mettre à jour le dictionnaire en ajoutant des «lignes», en en modifiant...

... et en en supprimant

Dict — itération et type

```
for key in ages:
    print(key)      # Alex
                   # Emelyne
                   # Victor
```

Une itération avec un simple for sur le dictionnaire revient à itérer à travers les clés

```
for key, value in ages.items():
    print(f"{key} -> {value}")

    # Alex -> 11
    # Emelyne -> 9
    # Victor -> 7
```

On peut itérer sur une «vue» du dictionnaire comme une séquence de paires, et chaque itération livre à la fois la clé et la valeur associée

```
ages: dict[str, int] = ...
```

Pour définir le type d'un dictionnaire, on indique à la fois le type des clés et le type des valeurs

Dict — defaultdict

```
from collections import defaultdict
```

```
hours_worked: list[tuple[str, int]] = [  
    ("Jane", 8),  
    ("John", 8),  
    ("Jane", 10),  
    ("Jane", 4),  
    ("John", 7),  
]
```

```
totals: defaultdict[str, int] = defaultdict(int)
```

```
for name, hours in hours_worked:  
    totals[name] += hours
```

```
print(totals["Jane"]) # 22  
print(totals["Mary"]) # 0
```

Un defaultdict assigne une valeur par défaut à une nouvelle clé qu'il ne connaît pas.

Exemple: une liste de combien d'heures 2 personnes ont travaillé. On demande les totaux par nom

On crée un defaultdict avec comme valeur par défaut `int()`, donc 0, obtenu via l'argument `int` de `defaultdict`

En accumulant les totaux, pas besoin de prendre en compte le cas où la clé n'est pas encore connue

Le dict répond (et insère) 0 s'il ne connaît pas la clé, par exemple ici pour Mary.

Cours 2

Structures de données:
tuple, list, set, dict, **dataclass**

Données simples: les *dataclasses*

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class Person:
```

```
    first_name: str
```

```
    last_name: str
```

```
    height: float
```

```
    num_children: int = 0
```

```
@property
```

```
def full_name(self) -> str:
```

```
    return f"{self.first_name} {self.last_name}"
```

```
p = Person("Jean-Philippe", "Pellet", 1.79, 3)
```

```
p.num_children += 1
```

```
p2 = Person("Zoé", "Zufferey", 1.72)
```

```
print(p2.full_name) # Zoé Zufferey
```

Une classe déclarée ainsi avec l'annotation @dataclass est conçue pour représenter des données de manière flexible et convient bien à une modélisation de base d'un problème

Chaque attribut avec son type est indiqué dans le corps de la classe, de façon indentée. En Python, tout est public (pas de concept de private ou protected)

On utilise la notation standard pour les méthodes pour définir des attributs dérivés, avec l'annotations @property

*On crée des instances comme pour un tuple nommé...
... et, cette fois, on a le droit de tout modifier.*

*On peut indiquer facilement des valeurs par défaut pour les attributs...
... ainsi que des attributs/propriétés dérivés*

Plus sur les méthodes

```
from dataclasses import dataclass
import math
```

```
@dataclass(order=True)
```

```
class Point:
```

```
    x: float
```

```
    y: float
```

```
    z: float
```

```
    def distanceTo(self, other: 'Point') -> float:
```

```
        dx = self.x - other.x
```

```
        dy = self.y - other.y
```

```
        dz = self.z - other.z
```

```
        return math.sqrt(dx * dx + dy * dy + dz * dz)
```

```
p1 = Point(0, 0, 0)
```

```
p2 = Point(1, 1, 0)
```

```
print(p1.distanceTo(p2)) # 1.4142135623730951
```

Les méthodes sont des fonctions liées à une classe, qui reçoivent automatiquement comme premier argument l'objet sur lequel la méthode est appelée

Cet exemple modélise un point en 3 dimensions et déclare une méthode pour calculer la distance à un autre point donné, passé en paramètre. order=True fait en sorte que des instances soit comparables avec < ou >

Une méthode doit toujours définir self comme premier argument, ce qui permet d'accéder aux attributs (ressemble à this dans d'autres langages. D'autres arguments peuvent être définis; le type peut être spécifié (ici entre '...' parce que faisant référence à au type de la classe en cours de définition)

Deux points sont créés

L'appel de la méthode ne mentionne pas explicitement self

Programmation orientée objet en Python

- **Les classes peuvent hériter d'autres classes**
 - ✦ Héritage multiple possible, comme en C++
- **Le pendant du constructeur est la méthode spéciale `__init__`**
 - ✦ Elle est générée automatiquement pour les dataclasses
- **Toutes les méthodes doivent déclarer *self* comme paramètre**
 - Elles font référence aux attributs/champs et appellent d'autres méthodes via ce *self*
- **Rien n'est privé, tout est public: notion de l'encapsulation basée sur conventions**
 - Un attribut qui commence par `_` est considéré comme «plutôt privé», à ne pas toucher
 - Un attribut qui commence par `__` est considéré comme «très privé»

Résumé du cours d'aujourd'hui



- **Les tuples sont définis avec des ()** — voire sans
 - ♦ Immuables, éléments potentiellement hétérogènes
- **Les listes sont définies avec des []**
 - ♦ Hautement flexibles; en général, contenant des éléments d'un même type
- **Les sets et les dicts sont définis avec des { }**
 - ♦ Les éléments (pour les sets) ou les clés (pour les dicts) sont uniques
 - * Pas de restrictions sur ce que peuvent être les valeurs des dicts
 - ♦ Les *defaultdicts* sont pratiques pour fournir une valeur par défaut
- **Les compréhensions de listes et de sets rendent certaines transformations faciles et concises**
- **Pour modéliser des types plus complexe, les dataclasses sont très pratique** (il y a aussi NamedTuples, immuables, et les classes normales)

Autoévaluation — objectifs



- **Je suis capable de/d'...**
 - ✦ déclarer et manipuler des tuples
 - ✦ déclarer et manipuler des listes en utilisant [] pour récupérer ou mettre à jour **un** (avec un index simple) ou **des** (avec du slicing) élément(s)
 - ✦ créer des listes dérivées avec la syntaxe des compréhensions de listes
 - ✦ déclarer et manipuler des sets
 - ✦ déclarer et manipuler des dicts, repérer quand un defaultdict aurait du sens
 - ✦ itérer à travers ces structures de données et convertir de l'une à l'autre
 - ✦ insérer les types de ces structures de données
 - ✦ argumenter pourquoi je choisis plutôt un tuple, une list ou un set pour résoudre un problème donné
 - ✦ déclarer et manipuler des *dataclasses*
 - ✦ énoncer les limites des uns et des autres

Pour avoir le vérificateur de type dans les notebooks

- Dans VS Code, installez **l'extension Mypy de Matan Grover** (en plus de celle, déjà installée, de Microsoft)
- Éditez le fichier **.vscode/settings.json** en tapant ceci dans un terminal VS Code:
`code .vscode/settings.json`
- Cela doit ouvrir un fichier déjà existant. Ajoutez-y ces lignes:

```
"mypy.checkNotebooks": true,  
"mypy.mypyExecutable": "${workspaceFolder}/venv/bin/mypy",  
"mypy.dmypyExecutable": "${workspaceFolder}/venv/bin/dmypy",
```

- *Alternativement, retéléchargez setup.sh et refaites-le tourner une fois*