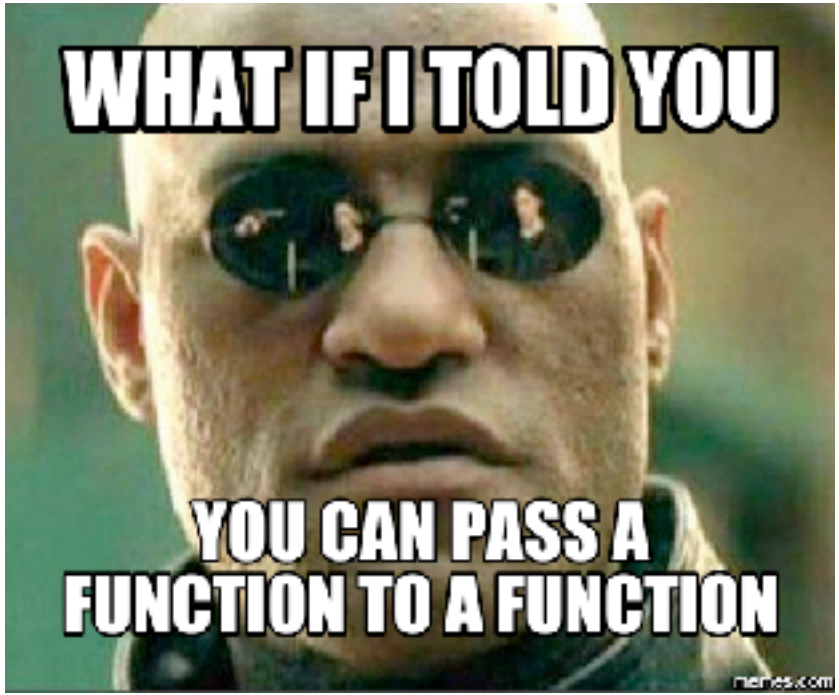



# Information, Cacul et Communication

Partie Programmation

Cours 12 : Compréhension de listes
Fonctions d'ordre supérieur
Lambda

08.12.2023 Patrick Wang

# Au sommaire cette semaine



1. Compréhension de listes/dictionnaires
2. Fonctions d’ordre supérieur
3. Lambda

# Au sommaire cette semaine



1. Compréhension de listes/dictionnaires
2. Fonctions d’ordre supérieur
3. Lambda

# 1. Compréhension de listes

## Retour sur la série 11

- Nettoyage du fichier nat2018.csv

```
boys: List[DataEntry] = []
girls: List[DataEntry] = []

with open('nat2018.csv', 'r', encoding='utf=8') as f:
    lines = [line for line in f.read().splitlines() if 'XXXX' not in line and '_PRENOMS_RARES' not in line]
    for line in lines[1:]:
        gender, name, birth_year, count = line.split(';')
        if gender == '1':
            boys.append(DataEntry('M', name, int(birth_year), int(count)))
        else:
            girls.append(DataEntry('F', name, int(birth_year), int(count)))
```

- Utilisation d'une compréhension de liste pour générer «rapidement» une liste

# 1. Compréhension de listes

## Retour sur la série 11

- Analyse du prénom CAMILLE

```
def find_camilles(boys: List[DataEntry], girls: List[DataEntry]) -> int:
    boys_camille: Dict[int, int] = {boy.birth_year: boy.count for boy in boys if boy.name == 'CAMILLE'}
    girls_camille: Dict[int, int] = {girl.birth_year: girl.count for girl in girls if girl.name == 'CAMILLE'}
    for birth_year in sorted(boys_camille.keys()):
        if birth_year in girls_camille.keys():
            if boys_camille[birth_year] < girls_camille[birth_year]:
                return birth_year
```

- Utilisation d'une compréhension de dictionnaire pour générer «rapidement» deux dictionnaires

# 1. Compréhension de listes

## Syntaxe de base – Syntaxe avec filtre

- Pour créer une liste par compréhension :

```
comp_list = [expression for val in iterable]
```

Expression qui définit les éléments de la liste par compréhension.
Peut se servir (ou pas) de `val`

Élément de l'itérable, à chaque itération

Objet itérable, assimilable à une liste

- Avec filtre :

```
comp_list = [expression for val in iterable if bool_expr]
```

Filtre des valeurs ajoutées à la liste

# 1. Compréhension de listes

## Retour sur la série 11

```
boys: List[DataEntry] = []
girls: List[DataEntry] = []

with open('nat2018.csv', 'r', encoding='utf=8') as f:
    lines = [line for line in f.read().splitlines() if 'XXXX' not in line and '_PRENOMS_RARES' not in line]
    for line in lines[1:]:
        gender, name, birth_year, count = line.split(';')
        if gender == '1':
            boys.append(DataEntry('M', name, int(birth_year), int(count)))
        else:
            girls.append(DataEntry('F', name, int(birth_year), int(count)))
```

Toutes les lignes du fichier nat2018.csv

Filtre à appliquer au moment de l'ajout d'éléments

- `lines: List[str]`, contient toutes les lignes de nat2018.csv sauf celles avec XXXX ou _PRENOMS_RARES

# 1. Compréhension de dictionnaires

## Retour sur la série 11

```
def find_camilles(boys: List[DataEntry], girls: List[DataEntry]) -> int:
    boys_camille: Dict[int, int] = {boy.birth_year: boy.count for boy in boys if boy.name == 'CAMILLE'}
    girls_camille: Dict[int, int] = {girl.birth_year: girl.count for girl in girls if girl.name == 'CAMILLE'}
    for birth_year in sorted(boys_camille.keys()):
        if birth_year in girls_camille.keys():
            if boys_camille[birth_year] < girls_camille[birth_year]:
                return birth_year
```

Clés du dictionnaires

Valeurs associées aux clés

- Attention !
  - Liste par compréhension : [ ... ] ; Tuple par compréhension : ( ... )
  - Dictionnaire par compréhension : { k: v ... }
  - Ensemble par compréhension : { v ... }

# 1. Compréhension de listes

## Syntaxes plus complexes

- Il est possible de combiner plusieurs itérables pour créer une structure de données dont les éléments sont construits à partir de deux valeurs

```
some_tuples = [(odd, even) for odd in range(0, 10, 2) for even in range(1, 10, 2)]
'''
# Complètement équivalent à :

some_tuples = []
for odd in range(0, 10, 2):
    for even in range(1, 10, 2):
        some_tuples.append((odd, even))
'''
```

# 1. Compréhension de listes

## Syntaxes plus complexes

- Il est possible de combiner plusieurs itérables pour créer des listes multi-dimensionnelles.

```
multiplications = [[factor * n for factor in range(1, 11)] for n in range(2, 11)]
'''
# Complètement équivalent à :

multiplications = []
for n in range(2, 11):
    table_n = []
    for factor in range(1, 11):
        table_n.append(factor * n)
    multplications.append(table_n)
'''
```

# Au sommaire cette semaine



1. Compréhension de listes/dictionnaires
2. Fonctions d’ordre supérieur
3. Lambda

# 2. Fonctions d'ordre supérieur

## Quelques considérations initiales

- Quels sont les types de variables que nous connaissons ?
  - Types scalaires: `int`, `float`, `bool`, `str`, ...
  - Collections : `List`, `Tuple`, `Dict`, `Set`, ...
  - Dataclasses
- On parle parfois de ***first class citizens*** ou de ***first class*** objects, i.e. :
  - ces objets peuvent être stockés dans une variable ou une collection
  - ces objets peuvent être passés en argument d'une fonction
  - ces objets peuvent être retournés par une fonction
  - ces objets sont comparables entre eux
  - ...

# 2. Fonctions d'ordre supérieur

## En Python, les fonctions sont aussi des first-class objects!

- Fonctions de première classe : qu'est-ce que ça peut bien vouloir dire ?
  - Stockable dans une variable :

```
afficher = print
afficher("Hello!")
```

  - Passé comme argument à une fonction :

```
def plus_2(n: int) -> int:
    return n + 2

def squared(f: Callable[[int], int], n: int) -> int:
    return f(f(n))

plus_4 = squared(plus_2)
print(plus_4(1))
```

Liste des types des paramètres de la fonction

Type de retour de la fonction

# 2. Fonctions d'ordre supérieur

## Définitions et quelques exemples

- Une fonction d'ordre supérieur (higher-order function, HOF) sont des fonctions qui prennent comme argument une autre fonction
- Comme dans l'exemple précédent, nous pouvons créer nos propres fonctions d'ordre supérieur
- Mais certaines sont définies par Python et disponibles :
  - `filter(f, list)`: filtre les éléments d'une liste grâce à une fonction (si `f` renvoie `True`, alors l'élément est gardé)
  - `map(f, list)`: applique la fonction `f` à tous les éléments de la liste

# 2. Fonctions d'ordre supérieur

## Retour sur la série 11

- Filtrer le contenu de nat2018.csv en retirant les lignes qui contiennent `'XXXX'` ou `'_PRENOMS_RARES'`

```
def filtering_lines(line: str) -> bool:
    return 'XXXX' not in line and '_PRENOMS_RARES' not in line

with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(filtering_lines, lines))
print(lines[:10])
```

Ne pas oublier de convertir le tout en une liste (valable aussi pour `map()`)

# Au sommaire cette semaine



1. Compréhension de listes/dictionnaires
2. Fonctions d'ordre supérieur
3. Lambda

# 3. Lambda

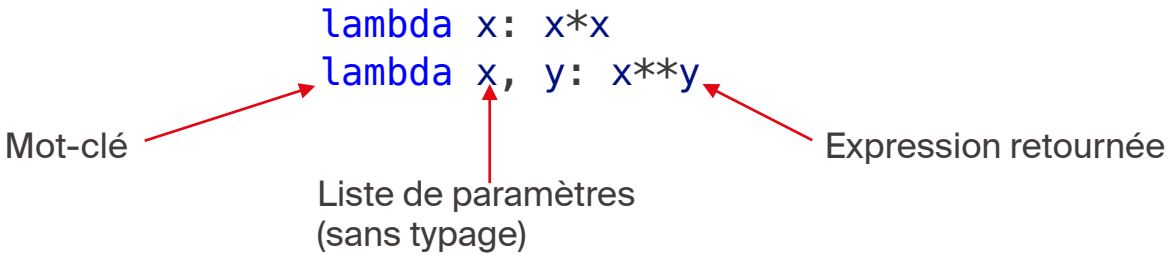
## Introduction des lambda

- Les fonctions lambda (ou fonctions anonymes) sont des fonctions (très courtes) qui n'ont pas de nom !
- À quoi ça peut bien servir ?
- Comment je fais pour appeler une fonction qui n'a pas de nom ?

# 3. Lambda

## Quelques exemples et syntaxe

- Pour définir une fonction lambda:




- Pour les appeler :

```
f = lambda x: x**x
g = lambda x, y: x**y
f(2)
g(2, 3)
```

# 3. Lambda

## Utilité avec les fonctions d'ordre supérieur

- C'est un peu embêtant de devoir définir `filtering_lines()` alors que cette fonction n'est utilisée qu'une seule fois...

```
def filtering_lines(line: str) -> bool:
    return 'XXXX' not in line and '_PRENOMS_RARES' not in line

with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(filtering_lines, lines))
print(lines[:10])
```

# 3. Lambda

## Utilité avec les fonctions d'ordre supérieur

- C'est un peu embêtant de devoir définir `filtering_lines()` alors que cette fonction n'est utilisée qu'une seule fois...

```
def filtering_lines(line: str) -> bool:
    return 'XXXX' not in line and '_PRENOMS_RARES' not in line

with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(filtering_lines, lines))
print(lines[:10])
```

# 3. Lambda

## Utilité avec les fonctions d'ordre supérieur

- C'est un peu embêtant de devoir définir `filtering_lines()` alors que cette fonction n'est utilisée qu'une seule fois...

```
def filtering_lines(line: str) -> bool:
    return 'XXXX' not in line and '_PRENOMS_RARES' not in line

with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(filtering_lines, lines))
print(lines[:10])
```

# 3. Lambda

## Utilité avec les fonctions d'ordre supérieur

- C'est un peu embêtant de devoir définir `filtering_lines()` alors que cette fonction n'est utilisée qu'une seule fois...

```
with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(lambda x: 'XXXX' not in x and '_PRENOMS_RARES' not in x, lines))
print(lines[:10])
```

# 3. Lambda

## Utilité avec les fonctions d'ordre supérieur

- C'est un peu embêtant de devoir définir `filtering_lines()` alors que cette fonction n'est utilisée qu'une seule fois...

```
with open('cours11-data/nat2018.csv', 'r', encoding='utf-8') as f:
    lines = f.read().splitlines()
lines = list(filter(lambda x: 'XXXX' not in x and '_PRENOMS_RARES' not in x, lines))
print(lines[:10])
```

# Bilan

- Compréhension de listes, dictionnaires, ensembles
  - C'est un peu du «syntactic sugar» mais ça peut être utile d'avoir vu ça une fois en cours pour s'en souvenir les prochaines fois que vous verrez ça
- Fonctions d'ordre supérieur et lambda:
  - C'est du contenu un peu avancé
  - Mais les utilisations sont plus courantes que ce qu'il n'y paraît
    - Filtres des éléments d'une liste
    - Applications d'une fonction à tous les éléments d'une liste
- Rendu du mini-projet sur Moodle en fin de semaine !