ICC: Programmation

MX, Cours 10, 22 novembre 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
- Boucles pour exécuter du code plusieurs fois:
- Déclaration de fonctions avec type de retour et paramètres
- Utilisation de listes, sets, dictionnaires, tuples
- Déclaration de classes simples et de méthodes dans ces classes
- Création d'interfaces graphiques simples avec Tkinter
  - Fenêtres, Frames, geometry manager, widgets

Cours de cette semaine

Compréhensions de listes
Fonctions d'ordre supérieur
Types génériques
Lambdas

Liste dérivée et compréhensions de listes

Créez une liste de int qui indique la taille de chaque string issu d'une liste de strings donnée

```
words: List[str] = ["Elvis", "has", "left", "the", "building"]
size_of_words: List[int] = []
for word in words:
    size = len(word)
    size_of_words.append(size)

print(words)          # ['Elvis', 'has', 'left', 'the', 'building']
print(size_of_words)  # [5, 3, 4, 3, 8]
```

On prépare une liste initialement vide

Pour chaque mot de la liste, on obtient sa longueur et on ajoute cette valeur à notre nouvelle liste

Compréhension de liste:

```
size_of_words = [len(word) for word in words]
```

«une liste formée par le résultat de l'expression *len(word)* pour chaque mot de la liste *words*»

Compréhensions de listes

```
[expression for x in container]
```

range(), liste, set, etc.

variable de boucle qui prendra successivement
toutes les valeurs des éléments contenus

expression (qui peut utiliser *x*) qui sera évaluée
pour créer chaque élément de la liste

- Moyen concis de définition de listes
  - Se lit plutôt de droite à gauche
- Plusieurs for enchaînables
- Possible de filter des itérations avec des if
- *Fonctionne aussi avec des sets*

Compréhensions de listes: exemples

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

variable non utilisée: 10 × la même valeur, 0

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

filtrage à la volée avec un *if*

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

moyen facile de faire des *range()* avec des floats

```
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']
```

deux *for-in* successifs: combinaisons des variantes, ici en plus avec un filtre

```
[[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]]
```

génération de listes de listes

Cours de cette semaine

Compréhensions de listes

Fonctions d'ordre supérieur

Types génériques

Lambdas

Fonctions: des valeurs comme les autres

- Nous avons utilisé des variables et paramètres pour stocker des valeurs «simples»
  - — int, float, bool
  - — List, Dict, Set, Tuple
  - — Des instances de nos propres classes (OilSpillEvent, Crane)
- On peut aussi les utiliser pour stocker et manipuler directement des fonctions

Démo

Des variables contenant des fonctions

```python
def square(x: int) -> int:
    return x * x

print(square(4))

au_carré: Callable[[int], int] = square
print(au_carré(4))


def square2(x: int) -> int:
    return int(math.pow(x, 2))

def square3(x: int) -> int:
    return x ** 2

all_functions: List[Callable[[int], int]] = [square, square2, square3]
for f in all_functions:
    print(f(4))
```

Fonction «normale»; appel «normal»

Variable assignée avec comme valeur… la fonction elle-même — sans appel avec «*(arg)*» à la fin

Type de quelque chose qui «peut être appelé», comme une fonction

La fonction dans la variable se comporte et s'appelle comme une fonction «normale»

On peut par exemple mettre 3 fonctions dans une liste pour les manipuler dans une boucle

Type: une liste de fonctions qui acceptent un int et retournent un int

Le type Callable[[...], ...]

- Ce qui peut être appelé avec «*(arguments)*»
- Plusieurs types pour les arguments, un type de retour
- Exemples:
  - `f: Callable[[], int] = …`
    - ➡ `f` s'appelle sans paramètre et retourne un `int`:
      `n: int = f()`
  - `f: Callable[[int], int] = …`
    - ➡ `f` s'appelle avec un `int` comme paramètre et retourne un `int`:
      `n: int = f(42)`
  - `f: Callable[[str, int], Set[str]] = …`
    - ➡ `f` s'appelle avec un string comme premier paramètre, un `int` comme second, et retourne un ensemble de strings:
      `strings: Set[str] = f("test", 5)`

Des fonctions comme paramètres

```python
def apply_twice(f: Callable[[int], int], value: int) -> int:
    return f(f(value))

print(apply_twice(square, 4))
```

⇒ `print(square(square(4))`

- Cette fonction accepte comme premier paramètre une fonction
- Elle l'utilise pour l'appliquer deux fois de suite à son argument
- Mais la fonction passée doit accepter un int et retourner un int… Trop restrictif

— Bon: les types sont optionnels en Python…

— … mais comment le faire tout en gardant des types?

Des types génériques

```python
T = TypeVar('T')

def apply_twice(f: Callable[[T], T], value: T) -> T:
    return f(f(value))

print(apply_twice(square, 4))
print(apply_twice(math.sqrt, 4.0))
```

OK avec T = int

OK avec T = float

- On déclare un type générique (T, U, R…), inconnu
- Le linter vérifie les contraintes de types
  - Ici, la fonction passée en paramètre accepte n'importe quoi (de type arbitraire T) mais doit retourner la même chose
  - `apply_twice` retourne le même type que le type de son second paramètre

Des fonctions qui retournent des fonctions

- Des fonctions peuvent accepter des fonctions
- Elles peuvent aussi retourner des fonctions!

Démo

```
def twice(f: Callable[[T], T]) -> Callable[[T], T]:
    def f_of_f(x: T) -> T:
        return f(f(x))
    return f_of_f

fourth_power = twice(square)
print(fourth_power(4))
```

La fonction *twice* accepte une fonction *f* et retourne une fonction qui est le résultat de la double application de *f*

Ici, *fourth_power* est donc une fonction qui accepte un int et qui retourne un int, et qui s'appelle comme n'importe quelle autre fonction

Cours de cette semaine

Compréhensions de listes
Fonctions d'ordre supérieur
Types génériques
Lambdas

Fonctions lambda (1/2)

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

plus_4 = twice(plus_2)
```

- Ici, on nomme la fonction `plus_2` pour la passer à `twice` comme argument
- Peut-on faire plus concis?
- ⇒ Fonction «anonyme», lambda

```
plus_4 = twice(lambda x: x + 2)
```

Fonctions lambda (2/2)

```
lambda x: x + 2
```

- Les fonctions lambda…
  - N'ont pas de nom
  - Déclarent une liste d'arguments (sans type) avant le deux-points
  - Ont à droite du deux-points une seule expression
  - Ont un return automatique

```
lambda x, y: x * y
```

Fonction qui accepte deux paramètres et retourne leur produit

```
lambda: 42
```

Fonction qui n'accepte aucun paramètre et retourne 42

```
lambda x:
    y = x + 2
    return y * y
```

Pas possible sur plusieurs lignes; utilisez une fonction normale.

Exemple: comme critère de tri

```python
from typing import List, Dict, Tuple

text = … # un texte à analyser

def freq_analysis(text: str) -> Dict[str, int]: ...
    ... # un dictionnaire qui relie chaque lettre à son nombre d'occurrences

frequencies: Dict[str, int] = freq_analysis(text)
frequencies_as_tuples: List[Tuple[str, int]] = list(frequencies.items())

sorted_frequencies: List[Tuple[str, int]] = sorted(
    frequencies_as_tuples, key=lambda t: -t[1]
)

print(sorted_frequencies)
```

Création du dictionnaire (cf. un cours précédent)

Conversion du dictionnaire en une liste de paires (non triées)

Tri de la liste de paires, en utilisant comme clé de tri le *second* élément de la paire (le nombre d'occurrences de la lettre). Sans cela, tri selon l'ordre alphabétique…

Critère de tri

```
sorted(frequencies_as_tuples, key=lambda t: -t[1])
```

- Le paramètre `key` est une fonction qui retourne, pour chaque élément de la liste à trier, la valeur selon laquelle trier cet élément
- Nous voulons trier selon le nombre d'occurrences de la lettre
- Ici, on retourne la case `1` du tuple, donc le nombre d'occurrences
- On l'inverse pour trier dans l'ordre décroissant

Résumé Cours 10

- Les fonctions peuvent aussi être stockées dans des variables
- Elles peuvent être manipulées comme les autres valeurs
  - Passées en paramètres
  - Retournées depuis d'autres fonctions
- Les types génériques peuvent servir à définir des fonctions d'ordre supérieur plus flexibles
- Les fonctions lambda permettent de déclarer des fonctions anonymes sans `def`

Séance d'exercices

BC 07-08: Salle à partager avec les gymnasiens jusqu'à 15h.

Merci!