ICC: Programmation

GC/MX, Cours 4, 11 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:
  - Boucle `while <condition>: ...`
  - Boucle `for i in range(...): …`
- Déclaration de fonctions avec type de retour et paramètres:
  - `def calculate_area(r: float) -> float: ...`
  - Mot clé `return` pour renvoyer une valeur depuis une fonction
  - Type spécial `None` pour indiquer qu'une fonction ne retourne rien

Répétition — Déclaration d'une fonction

- Déclarez une fonction qui retourne la longueur de l'hypoténuse dans un triangle rectangle en utilisant la longueur des deux cathètes.

Racine carrée de x: `math.sqrt(x: float) -> float`

```
import math
```

«J'utilise ce module externe dans mon code»

```
def calculate_hypothenuse(cathetus1: float, cathetus2: float) -> float:
    return math.sqrt(cathetus1 * cathetus1 + cathetus2 * cathetus2)
```

Priorité entre les opérateurs: * avant +
(comme algèbre standard)
⇒ pas de parenthèses requises ici

Valeurs par défaut des paramètres

«Si ces paramètres ne sont pas précisés lors de l'appel, utilise ces valeurs»

```
def say_hello(to: str = "John", n: int = 1) -> None:
    hello = " hello" * n
    print(f"Oh,{hello}, {to}!")


say_hello()       # valeurs pas précisée: on utilise to="John" et n=1

say_hello("John") # le premier paramètre est précisé, et n=1

say_hello(to = "John") # même effet ici

# say_hello(3)    # impossible, le premier paramètre est de type str

say_hello(n = 2)  # OK, car le paramètre est nommé

say_hello(n = 3, to = "James") # on peut réordrer les paramètres nommés
```

Cours de cette semaine

Listes

Motivation

- Listes:
  «Je veux stocker une série de valeurs en mémoire, mais je ne sais pas forcément à l'avance combien»

Avant de plonger: Rappel interpréteur

- Mode habituel de programmation:
  - Éditer un fichier
  - Sauvegarder
  - Faire tourner le code
- Pas pratique pour tester de petites choses
- En Python: interpréteur direct
  - Après chaque ligne ou bloc, exécution directe
  - Affichage de la valeur calculée par chaque ligne/bloc

Démo

Listes

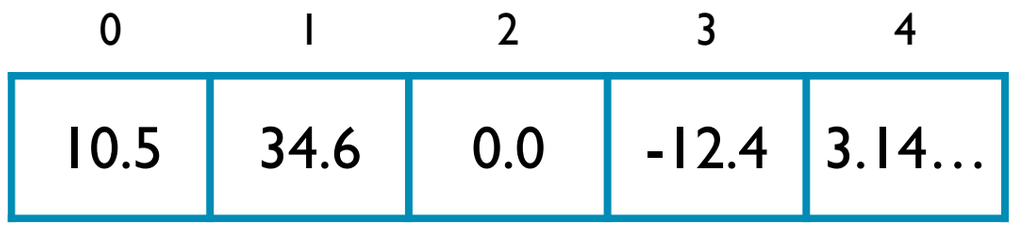
Variable simple = *une valeur* = *une «case»*

```
v: float = 42.5
v += 2.7
```

Liste = *séquence de valeurs (en général, du même type)* = *plusieurs «cases» numérotées*

```
# Possibilité 1: prédéfinition
numbers: List[float] = [10.5, 34.6, 0, -12.4, math.pi]

# Possibilité 2: construction dynamique
numbers = []
numbers.append(10.5)
numbers.append(34.6)
numbers.extend([0, -12.4, math.pi])
```



Exemples des semaines précédentes (I)

```
analyze_string("Bonjour")
analyze_string("programmation")
analyze_string("exercice")
```

```
data: List[str] = ["Bonjour", "programmation", "exercice"]
i: int = 0
while i < len(data):
    analyze_string(data[i])
    i += 1
```

Dans cette boucle, i commence à 0 et va jusqu'à len(data) – 1, donc vaudra 0, puis 1, puis 2

Le i^e élément de la liste *data*

«Pour chaque valeur dans cette liste, assigne-la à la variable *elem* puis exécute le corps de la boucle»

```
for elem in ["Bonjour", "programmation", "exercice"]:
    analyze_string(elem)
```

Exemples des semaines précédentes (II)

```
if operation == "+":
    result = number1 + number2
    print(f"{number1} + {number2} = {result}”)
```

```
if operation == "+" or "plus": # erroné
    ...
```

```
if operation == "+" or operation == "plus":
    ...
```

OK, mais long

```
if operation in ["+", "plus"]:
    ...
```

Yes! x *in* y teste si x est un élément que y contient

Rechercher dans une liste

Écrivez une fonction find *qui recherche la première position d'un élément dans une liste, ou retourne −1 s'il est inexistant*

```
odds = [1, 3, 5, 3, 7]
print(find(odds, 3)) # doit retourner 1
print(find(odds, 7)) # doit retourner 4
print(find(odds, 9)) # doit retourner -1
```

Démo

Rechercher dans une liste

```python
def find1(values: List[int], value: int) -> int:
    i = 0
    while i < len(values):
        if values[i] == value:
            return i
        i += 1
    return -1
```

Avec une boucle classique *while* de 0 à n – 1

Si la liste à la position *i* contient la valeur qu'on cherche, on renvoie *i*

Si on sort de la boucle sans avoir fait de *return i*, c'est que la liste ne contient pas la valeur cherchée

```python
def find2(values: List[int], value: int) -> int:
    for i, elem in enumerate(values):
        if elem == value:
            return i
    return -1
```

enumerate() permet de faire un *for-in* à deux variables: la première est l'index de liste, la seconde est la valeur à l'index donné

```python
def find3(values: List[int], value: int) -> int:
    if value in values:
        return values.index(value)
    else:
        return -1
```

On utilise le test avec *in* et ensuite la méthode fournie *index()*, qui marche sur les listes comme sur les strings *(cf. exercices d'il y a 2 semaines)*

Compter les occurrences

Écrivez une fonction qui compte les occurrences d'une certaine valeur dans une liste de int.

```
my_ints = [2, 2, 5, 6, 1, 6, 3, 2]
print(count_occurrences(2, my_ints)) # doit donne 3
print(count_occurrences(3, my_ints)) # doit donne 1
print(count_occurrences(4, my_ints)) # doit donne 0
```

Retourne un int (le nombre d'occurrences)

```
def count_occurrences(target: int, values: List[int]) -> int:
    num = 0
    for v in values:
        if v == target:
            num += 1
    return num
```

A besoin de la valeur à rechercher, *target*, et de la liste dans laquelle rechercher, *values*

Lorsqu'on trouve une valeur *v* dans la liste qui est égale à *target*, on incrémente notre compteur

On renvoie le nombre de fois qu'on a observé *target*

```
print(my_ints.count(2))
print(my_ints.count(3))
print(my_ints.count(4))
```

Désolé… :)

Autres méthodes utiles sur les listes

- `append(x)` — ajouter un élément
- `extend([x, y, z])` — ajouter plusieurs éléments
- `clear()` — tout effacer
- `insert(i, x)` — ajouter `x` à la position `i`
- `remove(x)` — supprime le premier `x`
- Slicing, non seulement pour faire des sous-listes *(cf. string et sous-string)*, mais pour aussi modifier la liste

Slicing

```
my_ints = [10, 20, 30, 40, 50, 60]
my_ints[0]           # 10

my_ints[0:2]         # [10, 20]

my_ints[:4]          # [10, 20, 30, 40]

my_ints[4:]          # [50, 60]
```

une seule position — déjà connu

une sous-liste de 0 (inclus) à 2 (exclu)

une sous-liste jusqu'à 4 (exclu)

une sous-liste depuis 4 (inclus)

```
my_ints[4:] = [55, 65]
my_ints              # [10, 20, 30, 40, 55, 65]

my_ints[4:] = []
my_ints              # [10, 20, 30, 40]

my_ints[0:0] = [0, 1, 2]
my_ints              # [0, 1, 2, 10, 20, 30, 40]

my_ints[0] = [0, 1, 2]
my_ints              # [[0, 1, 2], 1, 2, 10, 20, 30, 40]
```

remplacement d'une sous-liste

remplacement d'une sous-liste par liste vide ⇒ suppression d'éléments contigus

remplacement d'une sous-liste vide par une liste non-vide ⇒ insertion d'éléments contigus

[0:0] n'est pas la même chose que [0], qui désigne une case précise

Résumé Cours 4

- Les listes servent à stocker une série de valeurs dans une série de «cases» numérotées depuis 0
- On utilise les crochets, par exemple [i] pour accéder à la i^e valeur; on peut faire du slicing comme pour les strings avec la notation [start:end]
  - Le slicing sert aussi à modifier la liste
- Le *for-in* marche directement pour itérer sur les listes
  - Si index nécessaire: `for i, elem in enumerate(my_list): …`
- Les listes ont une série de méthodes prédéfinies pratiques