EPFL

ICC

Programmation

CGC/SIE, **Cours 4**

11 mars 2024

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

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 manipuler une série de valeurs en mémoire…

  — *mais je ne sais pas forcément à l'avance combien*

  — *en y faisant référence avec le même nom de variable»*

Listes

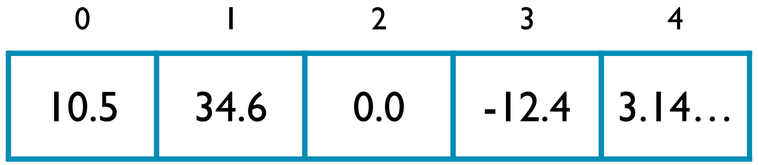
Variable simple = *une valeur* = *une «case»*

```
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

```
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

```
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

```
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

- `mylist.append(x)` — ajouter un élément
- `mylist.extend([x, y, z])` — ajouter plusieurs éléments
- `mylist.clear()` — tout effacer
- `mylist.insert(i, x)` — ajouter x à la position i
- `mylist.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. *On en reparle la semaine prochaine.*

Test case: Problème des 100 prisonniers

- 100 prisonniers dans des cellules numérotées de 0 à 99
- On prend les clés qu'on cache dans des boîtes numérotées de 0 à 99
- Chaque prisonnier peut ouvrir 50 boîtes (sans savoir ce qu'on fait les autres prisonniers)
- Si tous les prisonniers trouvent leur clé, ils peuvent tous partir.
  - Quelle meilleure stratégie, pour quelle probabilité de sortie?
    - ➡ Approche **naïve**: $1/2^{100}$
    - ➡ Approche **optimale**: environ 31%
  - https://www.youtube.com/watch?v=iSNsgj1OCLA

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Vous n'y croyez pas? Simulons ce problème!

Test case: Problème des 100 prisonniers

```python
from random import shuffle
n_prisoners = 100
n_attempts = 50

def simulate_one() -> bool:
    boxes = list(range(n_prisoners))
    shuffle(boxes)

    for p in range(n_prisoners):
        found_number = boxes[p]
        can_still_open = n_attempts - 1

        while can_still_open > 0 and found_number != p:
            found_number = boxes[found_number]
            can_still_open -= 1

        if found_number != p:
            return False

    return True
```

```python
n = 1000
successes = 0
for i in range(n):
    if simulate_one():
        successes += 1

print(successes / n)
```

Lance une simulation, et retourne *True* si les prisonniers ont réussi à sortir

Crée une liste de 0 à 99 et la mélange aléatoirement (reproduit le fait de cacher les clés)

Répète ceci pour chaque prisonnier avec *p* = numéro de cellule du prisonnier

On va voir le numéro derrière la boîte *p* et on se rappelle qu'on a le droit d'ouvrir encore 49 autres boîtes

Tant qu'on n'a pas trouvé la bonne clé et qu'on a encore le droit d'ouvrir d'autres boîtes, on le fait

Si le prisonnier n'a pas trouvé sa clé, c'est fichu pour tout le monde: on peut faire un *return False* directement depuis la boucle

Si on arrive ici, c'est que tous les prisonniers ont trouvé leur clé!

Pour avoir une estimation de probabilité, on répète l'expérience *n* fois et on compte le nombre de succès

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 *(on en reparle)*
- 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