Initiation au développement

3 — Modules et listes

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

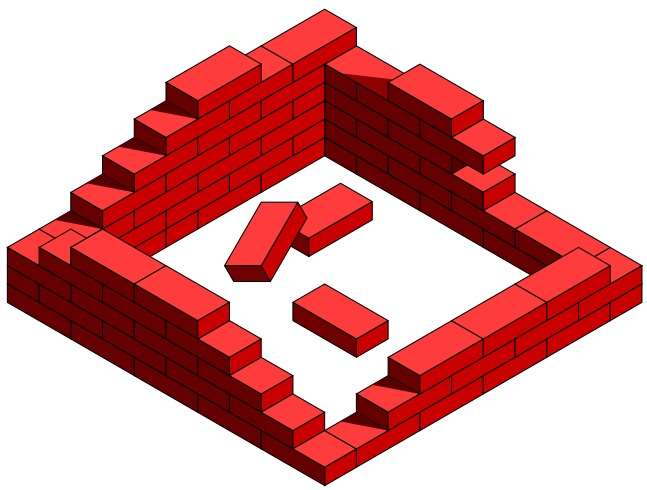
1. Les modules
2. Les listes
3. Listes et boucles `for`

Python pas à pas

Les modules

Modules

- Un **module** est un ensemble de fonctions déjà programmées, prêtes à être utilisées ;
- Python propose beaucoup de **modules de base** : (`math`, `time`, `random`, ...) ;
- On peut aussi en télécharger sur Internet ou en créer facilement ;
- Intérêt : recyclage de code !



Utilisation de modules

- Il y a trois façons d'utiliser une fonction `ma_fonction` d'un module **`mon_module`** :
  1. **`from mon_module import`** `ma_fonction` : on utilise alors normalement la fonction ;
  2. **`import mon_module`** : la fonction s'appelle `mon_module.ma_fonction` ;
  3. **`from mon_module import`** `*` : comme 1, mais on charge **toutes** les fonctions du module **`mon_module`** ;

Renommage de modules : `as`

L'import `from mon_module import *` est déconseillé !!!

- Il peut créer des problèmes : on peut écraser des fonctions possédant le même nom !
- Si le nom du module vous dérange, vous pouvez utiliser `as` ;
- Par exemple :

```
>>> import nombeaucouptroplong as nbtl
>>> x = nbtl.ma_fonction(12)
```

Un exemple

Exemple (module **math**)

```
import math

if __name__ == "__main__":
    print('pi = ', math.pi)
    print(math.cos(math.pi/4))
```

- **math** : les fonctions et les constantes mathématiques usuelles comme exp, cos, π, ... ;
- **random** : des fonctions pour faire des tirages au sort ;
- **time** : pour mesurer le temps, connaître l'heure, ou attendre un certain temps ;

Pourquoi “`if __name__ == '__main__'`”?

- Lorsque Python charge un module, il exécute tout le code exécutable qui s’y trouve ;
- La variable `__name__` contient le nom du module **`mon_module`**, à savoir :
  - `'mon_module'` s’il est chargé comme module ;
  - `'__main__'` s’il est chargé comme programme ;

Écrire votre code dans `main()` ou sous `if __name__ == '__main__'` permet de ne l’exécuter **que quand on écrit** `python3 mon_module.py`

Problèmes sans "__main__"

Essayons **import mon_module** avec les deux modules ci-dessous :

Sans "__main__"

```
# Fonctions
def super_fonction():
    pass

# Programme principal
n = input(
    "Entrez un nombre: "
)
# ...
```

Avec "__main__"

```
# Fonctions
def super_fonction():
    pass

# Programme principal
if __name__ == "__main__":
    n = input(
        "Entrez un nombre: "
    )
    # ...
```

“Et pourquoi **def** main() ?” . . . patience . . .

Créer ses propres modules

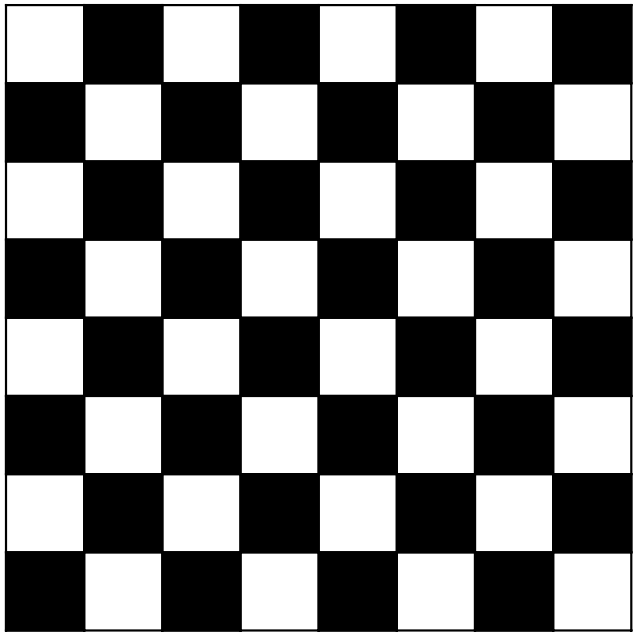
- Pour créer le module **`mon_module`**, il suffit d'écrire ses fonctions, constantes, ... dans un fichier `mon_module.py` ;
- Pour que Python le trouve, il doit être :
  1. dans un répertoire "standard" ;
  2. ou dans le même répertoire que votre **programme principal** ;
- Les règles vues pour nommer les variables s'appliquent aux modules ;

Python pas à pas

Les listes

Motivation

- Imaginons qu'on veuille programmer un jeu d'échecs en Python ;
- On a 64 cases à représenter :



- Par exemple par `case0, case1, ..., case63` ;
- . . . c'est possible, mais ni ergonomique ni facilement adaptable ;

Présentation

- La **liste** est une structure itérable ordonnée dont les données peuvent être de n'importe quels types ;
- On peut créer une liste par une affectation normale, où on met entre crochets et séparés par des virgules les différentes valeurs de la liste :

```
lst = [3, 'toto', 4.5, False]
```

- La **liste vide** s'obtient avec `list()` ou `[]` ;
- La syntaxe `lst[i]` donne accès au *i*-ème élément de la liste `lst` ;
- **Attention :** les indices commencent à 0 et non à 1 !

Exemple

```
>>> lst = [3, 'toto', 4.5, False]
>>> print(lst[1])
'toto'
>>> ['iut'] * 2
['iut', 'iut']
```

Affectations et listes

Il y a des subtilités ici, à bien travailler.

- Une liste est un objet **modifiable** ; on peut modifier ses valeurs, ses éléments, *etc* sans créer de nouvelle liste ;
- Ce n'est pas vrai pour les autres types qu'on a vus, notamment les chaînes de caractères ;
- On peut changer la *i*-ème valeur de `lst` avec une affectation classique : `lst[2] = 'titi'`

Affectations et listes

- Si `lst` est une liste, `lst2 = lst` associe au nom `lst2` la même liste (qui est modifiable) ;
- En conséquence, si on modifie ensuite `lst`, **on modifie aussi** `lst2` !

Exemple

```
>>> lst = [3, 'toto', 4.5, False]
>>> lst2 = lst
>>> lst2[1] = 'titi'
>>> print(lst)
[3, 'titi', 4.5, False]
```

[Visualisation sur PythonTutor]

- On peut obtenir une copie avec `list()` ;

[Visualisation sur PythonTutor]

Quelques opérations

- `len(lst)` renvoie la longueur de `lst` ;
- `x in lst` renvoie un booléen qui est **`True`** quand `x` est dans `lst` ;
- Le `+` crée une **nouvelle liste** qui est la concaténation de deux listes :

Exemple

```
>>> [3, 4, 7] + ['toto', 5]
[3, 4, 7, 'toto', 5]
```

- Si on multiplie une liste par un entier *n*, cela crée une **nouvelle liste** où l'ancienne est répétée *n* fois :

Exemple

```
>>> lst = [3, 4, 7]
>>> lst2 = lst * 3
>>> print(lst2)
[3, 4, 7, 3, 4, 7, 3, 4, 7]
```

Comparaison de listes

- Les opérateurs de comparaisons fonctionnent pour les listes :
  - `lst1 == lst2` vaut **`True`** seulement si les deux listes ont exactement le même contenu dans le même ordre ;
  - `lst1 != lst2` fait le contraire de l'opérateur précédent ;
  - les opérateurs `<`, `<=`, `>`, `>=` fonctionnent comme pour les chaînes de caractères **mais les types doivent être comparables** ;

Exemple

```
>>> [1, 2] < [3, 4]
True
>>> [1, 2] < ["a", "b"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int'
and 'str'
```

Autres opérations

- Les listes sont des **objets** disposant de **méthodes** : des fonctions rattachées au type `list` ;
- La syntaxe permettant d'utiliser ces méthodes sur une liste `lst` est `lst.methode()` ;
- **Attention :** en général, cela **modifie** la liste ;
- Quelques exemples :
  - `lst.append(x)` ajoute la valeur de `x` à la fin de `lst` ;
  - `lst.insert(i, x)` insère la valeur de `x` avant la position `i` la fin de `lst` ;
  - `lst.extend(lst2)` ajoute tous les éléments de `lst2` à la fin de `lst` ;
  - `lst.pop()` supprime le dernier élément de `lst` et renvoie sa valeur ;
  - `lst.pop(i)` supprime le *i*-ème élément de `lst` et renvoie sa valeur ;
  - `lst.remove(x)` supprime la première occurrence de `x` dans `lst` ;

Python pas à pas

Listes et boucles `for`

Structure itérable

- On a vu la fois passée ce qu'étaient les structures itérables ;
- On peut changer un itérable en la liste, dans l'ordre, de ses éléments avec la fonction `list()` ;
- C'est ce qu'on fait pour obtenir de façon explicite ce que contient un `range` :

Exemple

```
>>> range(0, 5)
range(0, 5)
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
```

Itérer sur une liste

- Une liste étant une structure itérable, on itère sur ses éléments comme sur ceux d'une chaîne ou d'un `range` :

Exemple

```
liste = [3, 5, 6, 14, -6, 121]
for element in liste:
    print(element)
```

- Si on a besoin des indices lors du parcours d'un itérable ordonné, on peut utiliser `range(len(iterable))`, vu que les indices vont de `0` à `len(iterable) - 1`

Exemple

```
lst = ['bon', 'jour', 'bonjour']
for i in range(len(lst)):
    print(i, lst[i])
```

Affectations dans un `for`

- Rappel : les affectations de “l’itérateur” ne fonctionnent pas comme on pourrait le croire !
- Exemple : si l’on veut ajouter 1 à tous les éléments d’une liste :

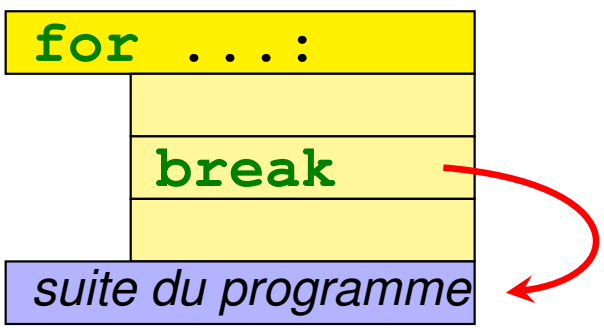
```
for element in liste:
    element += 1
```

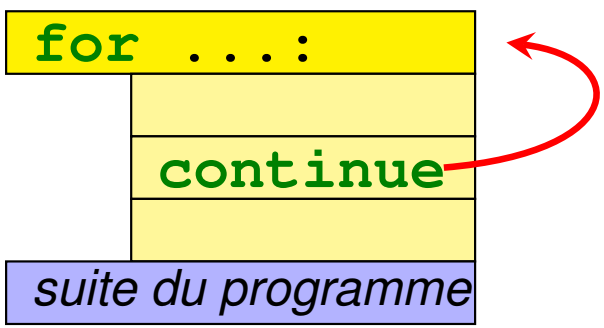
  ne marche pas ;
- On est **obligé** de recourir aux indices :

```
for i in range(len(liste)):
    liste[i] += 1
```

`break` et `continue`

- Il existe deux moyens pour interrompre une boucle :
  - `break` arrête la boucle et reprend donc l'exécution du programme après la boucle ;
  - `continue` arrête l'itération en cours (on "continue" à l'itération suivante de la boucle) ;





Un exemple avec **break**

- Exemple d'utilisation : si on cherche à savoir si une liste contient un nombre supérieur à 10, rien ne sert de poursuivre les recherches une fois qu'on en a trouvé un :

Exemple (trouver un élément $>$ 10)

```
# lst = ...
for element in lst:
    if element > 10:
        print(
            "J'ai trouvé", element, "dans la liste,"
            "qui est supérieur à 10"
        )
        break
```

Un exemple avec `continue`

- Exemple d'utilisation : n'afficher que les éléments d'une liste qui ne sont **pas** des chaînes :

Exemple (afficher une liste (sauf ses chaînes))

```
# lst = ...
for element in lst:
    if type(element) is str:
        continue
    print(element)
```

Éléments $\neq$ positions !!!

N'utilisez **jamais** `i` pour désigner un élément de la liste !!!

Exemple (correct)

```
for element in liste:
    # ... code sur des éléments
```

Exemple (correct)

```
for i in range(len(liste)):
    # ... code sur des positions
```

Exemple (incorrect)

```
for i in liste:
    # ... confusion éléments / positions!
    liste[i] = ... # plantage
```

Réservez les noms `i`, `j`, `k`, ... aux indices.