Initiation au développement

5 — Fonctions (2), tuples et boucle while

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

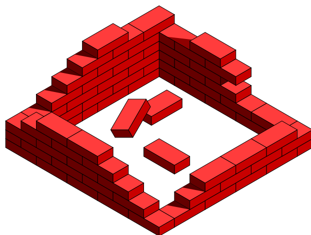
1. Clarifications sur les fonctions
2. Compléments sur les fonctions
   - Paramètres modifiables
   - Fonctions en argument
3. Les `tuple`s
4. La boucle **`while`**

Python pas à pas

Clarifications sur les fonctions

Intérêt des fonctions

- Les fonctions ont plusieurs intérêts ; entre autres :
  - pouvoir recycler du code ;
  - pouvoir exécuter du code **sans connaître les détails** ;



- Tout ce qu'il faut savoir doit être expliqué dans la docstring de la fonction :
  - que fait la fonction ?
  - que renvoie-t-elle et dans quel(s) cas ?
  - quels sont ses paramètres et les hypothèses à leur égard ?

Paramètres

- Les variables utilisées dans la fonction doivent être :
  - des variables locales déclarées dans la fonction ;
  - ou des paramètres ;

Il ne faut en aucun cas supposer l'existence d'autres variables car elles pourraient ne pas exister !

Exemple (à ne pas faire !)

```
def hamming():
    dist = 0
    for i in range(len(L)):
        dist += L[i] != M[i]
    return dist
```

Exemple (correct)

```
def hamming(L, M):
    dist = 0
    for i in range(len(L)):
        dist += L[i] != M[i]
    return dist
```

Paramètres et initialisation

- De la même manière, on suppose que les paramètres ont été initialisés avant d'appeler la fonction ;

Exemple (à ne pas faire !)

```python
def f(n):
    n = int(input("n = "))
    # ...
```

Exemple (correct)

```python
def f(n):
    # ...

n = int(input("n = "))
# code avec f(n)
```

Visibilité des fonctions

- Enfin, il faut que les fonctions soient “visibles” par Python pour qu’on puisse les récupérer dans d’autres modules ;
- Python acceptera sans broncher les exemples suivants, mais la fonction `g` ne sera jamais trouvable par les **`import`** :

Exemple (à ne pas faire !)

```
def f(...):
    # ...
    def g(...):
        # ...
```

Exemple (à ne pas faire !)

```
if __name__ == "__main__":
    def g():
        # ...
```

Rappelez-vous de la structure d’un programme :

1. tous les imports ; puis
2. toutes les fonctions ; et enfin :
3. le programme principal et / ou les tests, toujours sous `if __name__ == "__main__"` ;

Python pas à pas

Compléments sur les fonctions

Paramètres modifiables

- Rappel :

Exemple (appel de fonction)

```
def f(x):
    x = x + 1

if __name__ == "__main__":
    n = 3
    f(n)
    print(n)
```

- Le `x` de la fonction `f` est une variable locale de `f`, le `n` global n'est donc pas changé lors de l'appel à la fonction : cela affiche 3 ;

Paramètres modifiables

- Ce qu'on a évoqué la fois passée concernant les appels de fonctions et leurs paramètres est valable pour les paramètres non modifiables (entier, réel, booléen, chaîne, ...) ;
- Si un **paramètre** d'une fonction est modifiable, les méthodes de cet objet appelées dans la fonction le modifieront ;
- Si l'on agit sur un itérable modifiable, l'opérateur `[]` permettra également de modifier le paramètre ;

Paramètres modifiables

Exemple (appel de fonction sur une liste)

```
def ajoute(L, x):
    L.append(x) # on modifie L en ajoutant x


if __name__ == "__main__":
    lst = [4, 5]
    ajoute(lst, 7)  # L prend la valeur lst
    print(lst)      # affiche [4,5,7]
```

[Visualisation sur PythonTutor]

Variables locales et paramètres modifiables

Cela ne vaut que pour les **paramètres** modifiables ; les variables affectées dans la fonction restent locales.

Exemple (paramètre modifiable et affectation locale)

```
def f(L, x):
    L.append(x)
    L = list(range(3))
    L.append(x)

if __name__ == "__main__":
    lst = [4, 5]
    f(lst, 7)
    print(lst)
```

[Visualisation sur PythonTutor]

Fonctions en argument

- On peut passer une **fonction en argument** d'une autre fonction :

Exemple (filtrer les éléments d'un itérable)

```
def filtre(iterable, f):
    """Renvoie tous les éléments de l'itérable pour
     lesquels f(x) est vraie."""
    resultat = []
    for element in iterable:
        if f(element):
            resultat.append(element)
    return resultat

def est_pair(n):
    """Renvoie True si n est pair, False sinon."""
    return n % 2 == 0

if __name__ == "__main__":
    lst = filtre(range(10), est_pair)
    print(lst)
```

Fonctions en argument : exemple avec `min`

- Certaines fonctions standards de Python sont modifiables de la même manière :

Exemple (renvoyer la chaîne la plus courte)

```
>>> mots = ['a' * 8, 'b' * 4, 'c' * 2]
>>> min(mots)
'aaaaaaaa'
>>> min(mots, key=len)
'cc'
```

- Cela ne fonctionne bien sûr pas avec toutes les fonctions : consultez la documentation ;
- Exemples : `min`, `max`, `sorted`, ...

Python pas à pas

Les `tuples`

Les `tuples`

- Les `tuple`s sont des itérables équivalant à la notion mathématique de *n*-uplets ;

Exemple (déclaration de tuples)

```
>>> x = (4, 3, 1) # crée un tuple avec 3 entiers
>>> x = 4, 3, 1   # même résultat
>>> y = (4,)      # tuple avec un seul élément
```

- Pour créer un `tuple` avec **un seul élément**, il faut utiliser une virgule, sinon on obtient un entier ;
- Comme pour les listes et les chaînes :
  - on accède au *i*-ème élément avec `x[i]`, où les indices commencent à 0 ;
  - `len(x)` donne la longueur du tuple ;
  - `+` permet la concaténation, `* int` fonctionne aussi ;
- On peut changer un **itérable** en tuple à l'aide de la fonction `tuple` :

  `tuple(range(5))` ⇒ `(0, 1, 2, 3, 4)`

`tuple` vs `list`

Même si elles se ressemblent, `tuple` et `list` sont des structures complètement différentes

- La principale différence c'est que :
  - une `list` est **modifiable** ;
  - un `tuple` **n'est pas modifiable** ;
- Si `t` est un tuple, `t[0] = 3` produira donc une erreur ;
- Il n'y a pas de méthode `append()` pour les `tuple` ;
- ...
- En fait, un `tuple` ressemble plus à une chaîne qu'à une liste ;

Affectations multiples

- La notation des tuples permet d'affecter **simultanément** plusieurs variables comme ceci : `x, y, z = iterable`
- Il faut le **même nombre** d'éléments à gauche et à droite ;

Exemple

```
>>> x, y, z = [5, 6, 8]
>>> x, y, z = 5, 6, 8     # même résultat
>>> x, y = y, x           # échange les valeurs de x et y
>>> x, y = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> x, y, z = 1, 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
```

(illusion de) renvoyer plusieurs valeurs

- Comme une fonction peut renvoyer un `tuple`, on peut s'en servir pour renvoyer plusieurs valeurs :

Exemple

```
def div_euclidienne(a, b):
    """Renvoie le quotient et le reste."""
    return a // b, a % b

if __name__ == "__main__":
    q, r = div_euclidienne(14, 4)
    print('quotient=', q, ', reste=', r)
```

Python pas à pas

La boucle `while`

Boucle `while`

- On a déjà vu une instruction de boucle : la boucle `for` ;
- Il existe des situations dans lesquelles elle n'est pas adaptée ; principalement :
  - quand on veut faire une boucle infinie ;
  - quand on ne sait pas à l'avance de combien d'itérations on a besoin ;
  - quand on veut avoir plus de contrôle sur l'itérateur ;
- Dans ces cas-là, on se sert plutôt de la boucle `while` ;

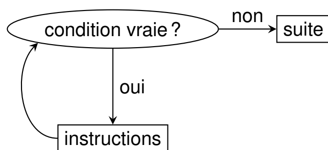
Structure de la boucle **while**

- La structure de la boucle **while** se présente comme suit :

Syntaxe du **while**

```
while condition:
    # instruction 1
    # instruction 2
    # ...
# suite
```

Organigramme



- Tous les exercices faits précédemment à l'aide d'une boucle **for** peuvent se réaliser aussi à l'aide d'une boucle **while** ;
- **Attention** : c'est à vous de vous assurer que la boucle se termine !

Deux exemples

Exemple (dix.py)

```
if __name__ == "__main__":
    nbr = int(input('Combien ? '))
    i = 1
    while i <= nbr:
        print('bonjour')
        i = i + 1
    print('Fini !')
```

Exemple (boom.py)

```
from time import sleep

if __name__ == "__main__":
    nbr = int(input('Combien de secondes ? '))
    i = nbr
    while i > 0:
        print(str(i) + '...')
        sleep(1)
        i = i - 1
    print('BOOOM !')
```

Nombre d'itérations non fixé

Le **while** a surtout un intérêt quand on ne sait pas à l'avance combien d'itérations seront nécessaires :

Exemple (forcer l'utilisateur à respecter les bornes)

```
if __name__ == "__main__":
    a = int(input('Nombre entre 1 et 10 : '))
    while a < 1 or a > 10:
        a = int(input('Erreur, entre 1 et 10 :'))
    print('Bravo, votre nombre est ' + str(a))
```

Exemple (lancer deux dés pour obtenir 12)

```
from random import randint

if __name__ == "__main__":
    des = 0
    nbr = 0
    while des != 12:
        des = randint(1, 6) + randint(1, 6)
        nbr = nbr + 1
    print(nbr, 'tirages pour faire 12')
```

Un exemple avec **break** (trouver.py)

- **break** et **continue** marchent aussi avec **while** :

Exemple (deviner un nombre)

```
from random import randint

if __name__ == "__main__":
    de = randint(1, 100)
    while True:
        a = int(input('Devinez un nombre (1-100) : '))
        if a == de:
            break
        if a < de:
            print('trop petit')
        if a > de:
            print('trop grand')
    print('Bravo !')
```

- **while True** boucle indéfiniment, la condition est toujours vérifiée ;
- **break** arrête les itérations quand le nombre est deviné ;

Boucles infinies

Si la condition de boucle est toujours vraie, et qu'il n'y a pas de `break` pour en sortir, le programme **reste bloqué en tournant indéfiniment dans la boucle !**

- Pour forcer l'arrêt d'un programme dans le terminal ou l'interpréteur : faire Ctrl+C ;
- Pour forcer la fermeture : faire Ctrl+D ;

Évaluation des conditions

- Quel programme effectue le moins de calculs ?

```
# L est une liste
for i in range(len(L)):
    print(L[i])
```

```
# L est une liste
i = 0
while i < len(L):
    print(L[i])
    i += 1
```

- La condition du **`while`** est évaluée à chaque début d'itération ...
- ... et `len(L)` est donc appelée à chaque évaluation ;
- Si `len(L)` ne change pas, on écrira la version avec **`while`** comme ceci :

```
# L est une liste
i = 0
n = len(L)
while i < n:
    print(L[i])
    i += 1
```

Modification d'itérables

Comme `len(L)` n'est évalué qu'une seule fois dans le **`for`**, il faut éviter d'utiliser un **`for`** pour supprimer des éléments d'une liste !

```
>>> lst = [1, 2, 3]
>>> for i in range(len(lst)):
...     lst.pop(i)
...
1
3
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: pop index out of range
```

Boucles imbriquées

- On peut utiliser des boucles dans des boucles, des boucles dans des boucles dans des boucles, ...;
- Cela arrive fréquemment quand on veut gérer des objets à deux dimensions, mais aussi dans d'autres situations (voir les algorithmes de tri dans le cours d'algorithmique) ;

Exemple (couples.py)

```
if __name__ == "__main__":
    for i in range(1, 5):
        for j in range(1, 4):
            print('(', i, ',', j, ')')
```

- Si vous n'êtes pas à l'aise avec ces boucles imbriquées, vous pouvez aussi remplacer la seconde boucle par un appel de fonction ;

Boucles imbriquées et `break`

- `break` fonctionne dans les boucles imbriquées, mais ne permet de sortir que de la boucle qui le contient directement !
- Si l'on veut sortir de toutes les boucles imbriquées directement, le plus simple est de placer son code dans une fonction et de remplacer `break` par `return` ;

Derniers mots sur les boucles

- `for` permet d'itérer sur les positions d'un itérable ou sur ses éléments ;
  - ⇒ si vous n'avez pas besoin des positions, forcez-vous à utiliser directement les éléments : le code sera plus lisible ;
- `while` est un peu plus compliqué à utiliser que `for` ;
  - ⇒ si vous pouvez vous en sortir avec un `for`, préférez cette solution-là :
    1. c'est plus simple à écrire ;
    2. on est certain de sortir de la boucle ;