

Variables

- Définition/affectation de variable : `a = ...`
À ne pas confondre avec `==` (comparaison de deux valeurs)

- Échanger les valeurs de 2 variables :

```
a, b = b, a
```

- Reste et quotient de la division euclidienne de `a` par `b` :

```
a % b # reste
a // b # quotient
```

- Un `float` est un nombre à virgule. On utilise un point au lieu d'une virgule pour les `float` :

```
pi = 3.14 # variable pi qui vaut 3.14
```

- Un `tuple` correspond aux n -uplets mathématiques :

```
t = (-1, 3.14) # définition d'un tuple
t[1] # récupère le 2ème élément de t (3.14)
```

Booléens et condition `if`

- Il y a 2 valeurs possibles pour un booléen : `True`, `False`

- Opérateurs de comparaison donnant un booléen :


```
==, != # égal, différent
<, <=, >, >= # inférieur, inférieur ou égal, ...
```

- Les opérateurs logiques :

```
and, or, not # et, ou, non
```

- Condition `if` :

```
if condition: # condition est un booléen
    instructions
```

Les instructions à l'intérieur d'un `if` (ou d'un `def`, `for`...) doivent être indentées (décalées) avec Tab ()

- Éviter les `if a == True` :

```
if a == True: # à éviter
if a: # équivalent préférable
```

- Condition `if` avec `else/elif` :

```
if c1:
    # à exécuter si c1 est True
elif c2:
    # à exécuter si c1 est False et c2 est True
else:
    # à exécuter si c1 et c2 sont False
```

Boucles

- Boucle `for` simple :

```
for i in range(10):
    print(i) # répété pour i=0, i=1, ..., i=9
```

La borne de fin est toujours **exclue** en Python (`range(n)` va jusqu'à `n - 1`).

- Boucle `for` avec début et fin :

```
for i in range(3, 10):
    print(i) # répété pour i=3, i=4, ..., i=9
```

- Boucle `for` avec début, fin et pas :

```
for i in range(3, 10, 2):
    print(i) # répété pour i=3, i=5, i=7, i=9
```

- Boucle `while` :

```
while c: # c est une condition (booléen)
    instructions # répété tant que c est True
```

Fonctions

- Définition d'une fonction :

```
def f(x, y): # x et y sont les arguments de f
    ... # instructions
    return ...
```

`return` arrête complètement la fonction : par exemple, un `return` dans une boucle la termine.

Une fonction peut avoir plusieurs `return`, mais un seul ne sera exécuté.

- Exemple d'utilisation de la fonction précédente : `f(3, 7)`.
Le corps de `f` sont exécutées avec `x = 3` et `y = 7`, et `f(3, 7)` est remplacé par la valeur du `return`.

Quand on définit une fonction, on utilise des noms de variables (et pas des valeurs) pour les arguments. On donne les valeurs aux arguments lorsqu'on veut utiliser la fonction.

Listes

- Exemple de définition d'une liste :

```
L = [3.14, 6, True] # liste contenant 3 éléments
```

- Taille d'une liste : `len(L)`

- Accès à l'élément d'indice `i` : `L[i]`

Les indices commencent à partir de 0 : le premier élément est `L[0]`, le dernier `L[len(L) - 1]` (qui est obtenu aussi avec `L[-1]`).

- Exemple de parcours de liste :

```
def appartient(e, L):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

Le `return False` est **après** la boucle `for`, pas dedans : c'est seulement lorsqu'on a parcouru la liste en entier sans trouver `e` qu'on est sûr que `e` n'est pas dans `L`.

- Ajout d'un élément `e` à la fin de `L`: `L.append(e)`

- `L[i:j]` extrait de `L` une sous-liste des indices `i` à `j - 1`.