	Python
test d'égalité	`==`
différent	`!=`
inférieur ou égal $\leq$	`<=`
division euclidienne	`//`
modulo	`%`
et	`and`
ou	`or`
négation	`not`

- On peut utiliser `if a % b == 0` pour savoir si `b` divise `a` (exemple : `if n % 2 == 0` pour savoir si `n` est pair).
- Ne pas confondre == (comparer deux valeurs, dans un `if` ou `while`) et = (modifier la valeur d'une variable).
- La variable modifiée est toujours à gauche du = : `a = b` modifie `a`.
- `L[i]` donne une erreur si `L[i]` n'existe pas :

```
L = []
L[0] = 0 # ERREUR !!!
L = [0] # faire ceci à la place
L = []
L.append(0) # ou ceci
```

- Ne pas écrire `if x == True` ou `if x == False` mais `if x` ou `if not x` (plus idiomatique).
- Si `L1` et `L2` sont des listes de tailles n_1 et n_2, `L1 + L2` donne en complexité $O(n_1 + n_2)$ une nouvelle liste contenant les éléments de `L1` suivis des éléments de `L2`.
- `n*L` duplique la liste `L` `n` fois (même chose que `L + L + ... + L`). Exemple : `[0]*4` donne `[0, 0, 0, 0]`.
- `[... for i in ...]` est une création de liste par compréhension. Par exemple, `[i**2 for i in range(5)]` donne `[0, 1, 4, 9, 16]` et est équivalent à :

```
L = []
for i in range(5):
    L.append(i**2)
```

- On peut aussi ajouter une condition dans une liste par compréhension : `[i**2 for i in range(5) if i % 2 == 0]` donne `[0, 4, 16]`.
- Opérations sur une matrice `M` (comme liste de listes) :

Python	Signification
`M[i][j]`	$m_{i,j}$ (élément ligne i, colonne j)
`M[i]`	$i$ème ligne
`len(M)`	nombre de lignes
`len(M[0])`	nombre de colonnes

- Créer une matrice $n \times p$ remplie de 0 :

```
M = [[0]*p for _ in range(n)]
```

Ou utiliser des boucles `for` et `append` :

```
M = []
for i in range(n):
    L = []
    for j in range(p):
        L.append(0)
    M.append(L)
```

Attention : le code ci dessous ne marche pas, car `M` a n fois la même ligne (modifier l'une modifie les autres).

```
M = []
L = []
for j in range(p):
    L.append(0)
for i in range(n):
    M.append(L) # M contient n fois la même liste L !!!
```

- Les types de base (`int`, `float`, `bool`...) sont copiés par défaut, contrairement aux `list` :

```
a = 3
b = a
b = 2 # ne modifie pas a
```

```
L1 = [3]
L2 = L1
L2[0] = 4 # modifie L1
```

De même lors du passage en argument d'une fonction :

```
def f(L):
    L[0] = 3
L1 = [2]
f(L1) # L1 est modifié
```

- 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]`).
- `L[i:j]` extrait de `L` une sous-liste des indices `i` à `j - 1`. On peut copier une liste avec `L[:]` ou `L.copy()`.
- Si `x` est une liste, un n-uplet, une chaîne de caractères ou un tableau numpy :
  - `x[i]` est le îème élément de `x`
  - `x[-i]` est le $i$ème élément en partant de la fin
  - `x[i:j]` extrait les éléments de `x` du $i$ème au $j$ème exclu (exemple : si)
  - `len(x)` est la taille de `x` En revanche, on ne peut pas modifier un n-uplet ou une chaîne de caractères (pas de `x[i] = ...` ou de `x.append(...)` dans ce cas).
- Éviter de faire plusieurs fois le même appel de fonction : stocker le résultat dans une variable à la place.
- Ne pas confondre indice et élément d'une liste : `for i in range(len(L))` parcourt les indices de `L` (`i` vaut 0, 1, 2, ..., `len(L) - 1`), alors que `for x in L` parcourt les éléments de `L` (`x` vaut `L[0]`, `L[1]`, `L[2]`, ..., `L[len(L) - 1]`).
- `for i in range(a, b, p)` parcourt les entiers de `a` à `b - 1` en allant de `p` en `p` (par défaut $a = 0$ et $p = 1$).
- Pour écrire une fonction récursive, il faut toujours un cas de base (qui ne fait pas appel à la fonction elle-même) et un cas récursif (qui fait appel à la fonction elle-même).