Types de bases et opérateurs associés

- `int` : nombres entiers de taille *non limitée*

`+`	addition	
`-`	soustraction	
`*`	multiplication	
`/`	division *décimale*	ex : `13/5` vaut `2.6`.
`**`	exponentiation	ex : `3**4` vaut 81
`//`	quotient dans la division euclidienne	ex : `13//5` vaut 2.
`%`	reste dans la division euclidienne	ex : `13%5` vaut 3. Tests de divisibilité.

- `float` : nombres en virgule flottante de taille limitée
  S'écrivent toujours avec le séparateur décimal `.`, mêmes opérateurs que sur les entiers (à l'exception de `//` et `%`).
- `bool` : deux valeurs possibles `True` et `False`

`not`	négation (unaire)	inverse la valeur de l'argument.
`or`	ou (binaire)	vaut `True` si au moins un des arguments vaut `True`.
`and`	et (binaire)	vaut `True` si les deux arguments valent `True`.

- `str` : chaines de caractères

`+`	concaténation	ex : `"Hello"+"World"` vaut `"HelloWorld"`
`*`	répétition	ex : `"Euh"*4` vaut `"EuhEuhEuhEuh"`
`len`	longueur	ex : `len("Python")` vaut 6
`[]`	ième caractères	numérotation depuis 0, ex : `"Hello"[1]` vaut `"e"`

Des conversions sont possibles entre ces divers types, par exemple, `int("34")` transforme la chaine de caractères `"34"` en l'entier `34`, `float(34)` transforme l'entier 34 en flottant `34.0`.

Comparaison

`>`	strictement supérieur	
`<`	strictement inférieur	
`>=`	supérieur ou égal	les symboles sont dans l'ordre de leur lecture
`<=`	inférieur ou égal	
`==`	égal	⚠ à ne pas confondre avec `=` utilisé pour l'affectation des variables
`!=`	différent	

Instructions conditionnelles

- Pour exécuter des `<instructions>` si une `condition` est vérifiée :

Syntaxe :

```
if <condition>:
    <instructions>
```

Exemple :

```
if a!=0:
    b = 1/a
```

- Pour exécuter `<instructions1>` si une `condition` est vérifiée et `<instructions2>` sinon :

Syntaxe :

```
if <condition>:
    <instructions1>
else:
    <instructions2>
```

Exemple :

```
if x < y:
    minimum = x
else:
    minimum = y
```

Pour imbriquer plusieurs `if ... else`, on utilise `elif`.

Boucles `while`

Les boucles `while` répètent un bloc d'instruction tant qu'une condition est vraie.

Syntaxe :

```
while <condition>:
    <instructions>
```

Exemple :

```
rep = ""
while rep!='O' and rep!='N':
    rep = input("O/N ?")
```

Boucles `for` avec `range`

- L'instruction `range` génère des entiers et peut prendre un, deux ou trois arguments :

`range(n)`	génère les `n` entiers de l'intervalle $[\![0; n-1]\!]$
`range(m,n)`	génère les entiers de l'intervalle $[\![m; n-1]\!]$
`range(m,n,s)`	génère les entiers de l'intervalle $[\![m; n-1]\!] \cap \{m + ks, s \in \mathbb{N}\}$

⚠ Dans les trois cas, la valeur `n` n'est pas prise.

- L'instruction `range` peut s'utiliser conjointement avec une boucle `for` pour créer une variable qui prendra les valeurs générées par le `range`, le bloc d'`<instructions>` qui suit est alors exécuté pour chaque valeur de la variable.

Syntaxe :

```
for <variable> in range(...):
    <instructions1>
```

Exemple :

```
for i in range(1,9):
    print(i) # va afficher 1, 2,... 8
```

Une boucle `for` permet donc en particulier de *répéter* un nombre donné de fois des instructions.

Fonctions

- Les fonctions sont des blocs d'instructions réutilisables (chaque appel à la fonction exécute son bloc d'instruction), leur définition commence par le mot clé `def` puis ont écrit le nom de la fonction puis la liste de ses arguments entre parenthèses (séparés par des virgules).
- Une fonction peut prendre zéro, un ou plusieurs arguments.
- Une fonction *peut* renvoyer un résultat à l'aide d'une instruction `return`.
- Exemple : une fonction à 3 arguments et qui renvoie une valeur

```
def discriminant(a,b,c):
    return b**2 - 4*a*c
```

- Exemple : une fonction à un argument et qui ne renvoie rien (elle produit un affichage)

```
def triangle(n):
    for i in range(1,n):
        print("*"*i)
```

⚠ Ne pas confondre `print` et `return`. La fonction `discriminant` ci-dessus *renvoie* un résultat, donc on pourrait écrire `d = discriminant(1,-11,30)` afin de récupérer dans `d` la valeur calculée. La fonction `triangle` ne renvoie rien, elle produit un affichage, il serait donc illogique d'écrire `t = triangle(4)`. Par contre `triangle(4)` fera appel à cette fonction et affichera un triangle de 3 lignes d'étoiles.