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.

Tranches

❶ Accès à un caractère par son indice
La notation `[i]` déjà rencontrée sur les chaines de caractères permet d'accéder au `i`-ème caractère d'une chaine où les caractères sont numérotés *à partir de 0*. Par exemple, si `exemple = "Un petit exemple"` :

U	n		p	e	t	i	t		e	x	e	m	p	l	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

alors : `exemple[0]` est `'U'`, `exemple[1]` est `'n'`, ...

❗ On remarquera que l'indice du dernier élément est *la longueur de la chaine moins 1*. La longueur s'obtenant avec `len`, ici on a par exemple `exemple[len(exemple)-1]` qui vaut `'e'`.

❷ Tranches
On peut aussi prendre une tranche en précisant dans les `[]` le début de la tranche (inclus) et sa fin (exclue) séparé par le caractère `:` si le début ou la fin sont absents alors ils correspondent respectivement au premier et au dernier indice. Par exemples :

- `exemple[3:8]` est `"petit"`
- `exemple[:2]` est `"Un"` (le début étant absent, on commence au premier caractère)
- `exemple[13:]` est `"ple"` (la fin étant absente, on termine au dernier caractère)

❸ Pas de progression
Une tranche peut prendre un troisième paramètre qui indique alors un *pas de progression*, par exemple si ce pas vaut 2, on ne prend qu'un caractère sur 2. D'autre part si le pas est négatif alors on progresse de la fin de la chaine vers le début. Par exemples :

- `exemple[4:10:2]` est `"ei "`
- `exemple[15:8:-1]` est `"elpmexe"`
- `exemple[::-1]` est `"elpmexe titep nU"` le pas étant négatif on progresse de la fin (absente donc dernier caractère) jusqu'au début (absent donc premier caractère).

Tuples

❶ Un tuple est une suite de valeurs repérées par leur indice (à la façon des caractères d'une chaine). Un tuple se note entre `()` et les valeurs sont séparées par des virgules. Par exemple `date = (2, "décembre", 1815)` est un tuple constituées de trois valeurs.

❷ On retrouve pour les tuples, la fonction `len`, l'accès au ième élément avec `[i]` et les tranches déjà vues sur les chaines de caractères.

❸ Les valeurs d'un tuple ne sont *pas modifiables* (comme les caractères d'une chaines), une tentative en ce sens produit un `TypeError`

❹ Un tuple peut être décompacté afin d'affecter chacune de ses valeurs à une variable. Par exemple `jour, mois, annee = date`.

Importation de fonctions

En Python, on peut importer des fonctions se trouvant dans d'autres modules, deux syntaxes sont possibles :

- `from <module> import <fonction>`, cela rend directement utilisable `<fonction>` dans la suite du programme. Par exemple la fonction racine carrée s'appelle `sqrt` et doit être importé depuis le module `math` avec `from math import sqrt` pour être utilisable.
- `import <module>`, dans ce cas, toutes les fonctions du module sont utilisables mais on doit préfixer leur nom par celui du module. Par exemple après un `import math` pour utiliser la fonction racine carrée, on doit écrire `math.sqrt`.

Listes

❶ Une liste est une suite de valeurs repérées par leur indice. Une liste se note entre `[` et `]` et les valeurs sont séparées par des virgules. Par exemple `premiers = [2, 3, 5, 7, 11, 13, 15]` est une liste. La liste vide est `[]`.

❷ On retrouve pour les listes, la fonction `len`, l'accès au ième élément avec `[i]` et les tranches déjà vues sur les chaines de caractères et les tuples.

❸ Les valeurs d'une liste, à la différence de celles d'un tuple,sont *modifiables*, on peut donc écrire `premiers[6]=17` afin que la liste ci-dessus devienne `premiers = [2, 3, 5, 7, 11, 13, 17]`.

❹ On peut ajouter un élément à une liste avec `append`, la syntaxe est `<liste>.append(<element>)`. Par exemple, après exécution de `premiers.append(19)` la liste ci-dessus devient `premiers = [2, 3, 5, 7, 11, 13, 17, 19]`.

❺ On peut retirer le dernier élément d'une liste avec `pop`, la syntaxe est `<liste>.pop()`. L'élément retiré est renvoyé par cette instruction et peut-être récupéré, ainsi `n = premiers.pop()` aura deux effets : supprimer 19 de la liste `premiers` et affecter cette valeur à `n`.

❻ Création de listes :

- en donnant explicitement ses éléments (comme la liste `premiers` ci-dessus).
- par répétition avec `*`, par exemple `[77]*10` est la liste constitué de 10 fois le nombre 77.
- par ajout successif, on part d'une liste vide et on ajoute (généralement à l'aide d'une boucle `for`) successivement avec `append` les éléments à la liste.
- par compréhension, à la façon dont on définit parfois les ensembles en mathématiques. Par exemple, `[i for i in range(50) if i%10==7]` est la liste `[7, 17, 27, 37, 47]` (les nombres entre 0 et 49 dont le reste dans la division euclidienne par 10 est 7).

Mutables et non mutables

En Python, on manipule des objets, certains sont *mutables* et d'autres non. Une variable est l'association entre une étiquette (le nom) et l'objet qu'il référence. Les listes de Python sont mutables au contraire de tous les autres rencontrés jusqu'ici (`int`, `float`, `bool`, `str`, `tuple`). Lorsqu'on modifie un objet mutable, cela modifie toutes ses *références*.

Cas non mutable :

```
n = 42
m = n
m = m + 1
# n vaut toujours 42
# m et n référencent deux objets diff
```

Cas mutable :

```
n = [12, 15]
m = n
m.append(17)
#m référence maintenant [12, 15, 17]
#m et n référencent le même objet
```

Les fonctions qui modifient un objet mutable le font généralement sans faire apparaître le signe `=`, par exemple `liste.pop()`.