

## VARIABLES ET AFFECTATION

En python, pour stocker des données en mémoire, on utilise **des variables**, auxquelles il faut donner un nom, qui les désigne au sein du programme. Concrètement, une variable représente un emplacement dans la mémoire de l'ordinateur.

Lorsque l'on donne une valeur à une variable, on dit que l'on **affecte** cette valeur à la variable. Pour cela, on utilise l'opérateur d'affectation, qui est le signe "=".

```
1 nb = 12          # Dans la variable nb, on a mis le nombre 12
2 nb = nb + 2     # Dans nb, il y a maintenant le nombre 14
3 mot = "bonjour"
4 # La variable mot contient la chaîne de caractères "boujour".
```

**Remarque :** Les chaînes de caractères doivent être encadrées par des guillemets.

Il existe différents **types de données**, dont voici les principaux :

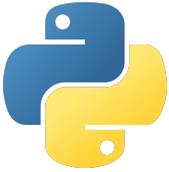
Types de données	Nom en python
Nombre entier	<code>int</code>
Nombre « à virgule »	<code>float</code>
Variable booléenne : vrai ou faux (True/False)	<code>bool</code>
Liste (ordonnée)	<code>list</code>
Chaîne de caractères (du texte...)	<code>str</code>

Ponctuellement, on pourra en rencontrer d'autres comme les ensembles non ordonnés (`set`) ou les dictionnaires (`dict`)...

## CONVERSION DE TYPE

On peut demander à python de convertir le type d'une variable. Il suffit d'utiliser la fonction associée au nom du type. Attention, le résultat de la conversion doit être stocké dans une variable.

```
1 ch = "42"       # Pour Python, ch contient le texte "42"
2 nb = int(ch)    # Pour Python, nb est le nombre entier 42
```



## LES ENTRÉES/SORTIES

- Pour écrire dans la console, on utilise la fonction `print()`.
- Pour demander à l'utilisateur d'entrer une chaîne de caractères, on utilise la fonction `input()`.



Le résultat de la fonction `input` est toujours une chaîne de caractères. Si elle doit être interprétée comme un nombre, il faut la convertir à l'aide des fonctions `int` ou `float`.

```
1 chaine_age = input("entrez votre age")
2 age = int(chaine_age)
3 date = 2017 - age
4 print("Vous êtes né en ", date)
```

## NOMBRES ET CALCULS

### Opérations valides avec tous les types de nombres

- Dans le langage Python, le séparateur décimal est le point : 1,3 se note 1.3
- Les opérations  $+$ ,  $-$ ,  $\times$  et  $\div$  sont représentées par les caractères  $+$ ,  $-$ ,  $*$  et  $/$ .
- La puissance se note «  $**$  ». Par exemple, pour calculer  $4^5$ , on utilise `4**5`.
- Python respecte les priorités opératoires usuelles.

### Opérations valides seulement avec des nombres entiers

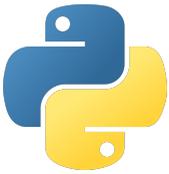
- On dispose de deux opérations liées à la division euclidienne.
  - L'opérateur `%` (on dit « modulo ») donne le reste d'une division euclidienne.
  - L'opérateur `//` (division entière) donne le quotient d'une division euclidienne.

Par exemple, la division euclidienne de 33 par 4 s'écrit  $33 = 4 \times 8 + 1$  donc :

$$33 \% 4 \rightarrow 1$$

$$33//4 \rightarrow 8$$

**Remarque :** L'opérateur `%` permet de tester si un nombre est divisible par un autre. Par exemple `n % 3 == 0` est vraie si et seulement si `n` est divisible par 3.



## INSTRUCTIONS CONDITIONNELLES

Une instruction conditionnelle permet d'effectuer un bloc d'instructions si et seulement si une certaine condition est vérifiée.

```
si condition(s) alors
|   Des instructions qui sont effectuées
|   si et seulement si la condition est réalisée
fin
```

On peut parfois souhaiter exécuter certaines instructions si une condition est vérifiée, et d'autres instructions dans le cas contraire :

```
si condition(s) alors
|   Suite d'instructions à effectuer
|   si la condition est vérifiée
fin
sinon
|   Suite d'instruction à effectuer
|   si la condition n'est pas vérifiée
fin
```

Voici comment cela se présente en Python, sur un exemple :

```
1 age = int(input("entrez votre age"))
2 if age >= 18 :
3     print("vous être majeur")
4 else :
5     temps = 18 - age
6     print("dans ", temps, "années, vous serez majeur" )
7
8 print("fin du programme")
```

### Remarque :

1. On peut ajouter à cette structure des blocs « sinon si » avec le mot clef **elif**.
2. La condition peut être n'importe quelle expression qui renvoie une valeur booléenne : Soit vraie, soit fausse.
3. La condition peut donc être simplement une variable booléenne.

## LES CONDITIONS EN PYTHON

---

Les opérateurs de comparaisons servent à réaliser des tests, qui peuvent servir de condition pour les instructions `if` et `while`.

Le résultat d'un test est une valeur **booléenne** : Vrai ou Faux.

opérateur de comparaison	signification
<code>==</code>	égal
<code>!=</code>	différent
<code>&lt;=</code>	inférieur ou égal
<code>&lt;</code>	strictement inférieur
<code>&gt;=</code>	supérieur ou égal
<code>&gt;</code>	strictement supérieur
opérateur d'appartenance	
<code>in</code>	appartient à



Le test d'égalité se fait avec l'opérateur « `==` » et non simplement « `=` », qui lui est l'opérateur d'affectation.

L'opérateur d'appartenance est souvent utile :

```
1 def EstVoyelle(lettre) :
2     if lettre in ["a","e","i","o","u","y"] :
3         return True
4     else :
5         return False
```

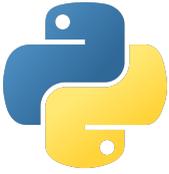
### PETIT POINT SUR LES OPÉRATIONS BOOLÉENNES

---

expression logique	rôle
<code>a and b</code>	vraie si <b>a</b> <b>et</b> <b>b</b> sont vraies
<code>a or b</code>	vraie si <b>a</b> <b>ou</b> <b>b</b> (ou les deux) sont vraies
<code>not(a)</code>	si <b>a</b> est vraie, <code>not(a)</code> est fausse, et inversement

Voici un exemple de fonction qui teste si un nombre `x` passé en paramètre appartient à l'intervalle `[1, 2[`.

```
1 def DansInter(x) :
2     if (x >= 1) and (x < 2) :
3         return True
4     else :
5         return False
```



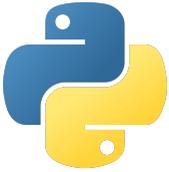
# LES FONCTIONS EN PYTHON

Fiche synthèse n° 4

Voici un exemple d'une fonction en Python qui prend en **paramètres** deux nombres, qui représentent les deux côtés de l'angle droit d'un triangle rectangle. Elle **renvoie** alors la longueur du troisième côté.

```
1 # On importe la fonction "racine carrée"
2 # depuis la bibliothèque "math"
3 from math import sqrt
4
5 def Pythagore(a,b) :
6     c = a*a + b*b
7     c = sqrt(c)
8     return c
9
10 # On peu maintenant utiliser notre fonction
11 hyp = Pythagore(3,4) # On affecte le résultat à une variable.
12 print(hyp)          # On obtient "5" dans la console
```

- Le mot clef **def** signale à Python que l'on définit une fonction ;
- **Pythagore** est le nom que l'on donne à notre fonction ;
- **a** et **b** sont les **paramètres** de notre fonction.
- Ne pas oublier les « deux points » en fin de ligne.
- Le **bloc d'instruction** qui correspond aux instructions effectuée par la fonction est « décalé à droite » par rapport au reste du programme.
- L'instruction **return** renvoie une valeur au programme principal et **arrête** l'exécution de la fonction.
- Il n'est pas obligatoire qu'une fonction contienne une instruction **return**, auquel cas la fonction ne renvoie rien au programme principale. (On parle alors de procédure)
- Si une fonction renvoie une valeur booléenne (**True** ou **False**), on peut s'en servir comme condition pour une instruction **if** ou **while**.



## IMPORTER UNE BIBLIOTHÈQUE

Il y a plusieurs façons d'importer une bibliothèque dans un programme Python.

```
1 import math
2 """ On accède alors aux fonctions du module math
3 en les préfixant par le nom de la bibliothèque. """
4 racine_deux = math.sqrt(2)
```

```
1 import math as m
2 """ On accède alors aux fonctions du module math
3 en les préfixant par m. """
4 racine_deux = m.sqrt(2)
```

```
1 from math import sqrt
2 """ La fonction sqrt est importée dans l'espace de noms
3 courant. On l'utilise alors directement. """
4 racine_deux = sqrt(2)
```

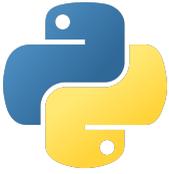
## DEUX BIBLIOTHÈQUES UTILES

1. Quelques fonctions utiles du module **math**

<code>sqrt()</code>	racine carrée d'un nombre
<code>log()</code>	logarithme népérien (ln)
<code>exp()</code>	fonction exponentielle
<code>sin()</code> , <code>cos()</code> , <code>tan()</code>	Sinus, cosinus, tangente d'un angle (en radian)
<code>floor()</code>	partie entière. <code>floor(3.9) → 3</code>
<code>pi</code>	une valeur approchée précise du nombre $\pi$

2. Quelques fonctions utiles du module **random**

<code>random()</code>	nombre aléatoire "réel" dans $[0; 1[$
<code>randint(a,b)</code>	nombre entier aléatoire entre <b>a</b> et <b>b inclus</b>



# LA BOUCLE TANT QUE...

---

Fiche synthèse n° 6

## RÉPÉTER UN BLOC SELON UNE CONDITION

---

Une boucle **while** permet de répéter un bloc d'instruction **tant qu'** une condition est vérifiée.

Voici un programme qui affiche les carrés des nombres entiers de 0 à 10.

```
1 n = 0
2 while n <= 10 :
3     carre = n*n
4     print(carre)
5     n = n + 1
```

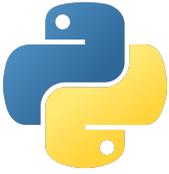
L'instruction **while** accepte n'importe quel test de condition, ou une variable booléenne, comme l'instruction **if**.

## LES PIÈGES À ÉVITER

---

- Ne pas oublier les « deux points » en fin de ligne. (comme pour **for**, **def** et **if**).
- Il ne faut pas oublier de modifier les paramètres de la condition dans la boucle, sans quoi le programme ne s'arrête jamais...
- Attention à l'indentation ! le programme suivant ne fonctionne pas car l'instruction « **n = n + 1** » **n'est pas dans la boucle**.

```
1 n = 0
2 while n <= 10 :
3     carre = n*n
4     print(carre)
5 n = n + 1 # Voilà le problème !
```



## RÉPÉTER UN BLOC UN NOMBRE DÉFINI DE FOIS...

Une boucle **for** permet de répéter un bloc d'instruction un nombre **prédéterminé** de fois.

Voici un programme qui affiche les carrées de nombres de 0 à 10, à mettre en parallèle avec celui de la fiche 6.

```
1 for i in range(0,11) : # La dernière valeur de i sera 10 !
2     carre = i*i
3     print(carre)
```

## LES PIÈGES



La borne supérieure de de la fonction **range** n'est pas atteinte. **range(0,6)** décrit les nombres 0, 1, 2, 3, 4, 5 mais **pas** 6.

1. Dans la boucle, on peut utiliser le compteur **i**, mais il ne faut pas le modifier !
2. Il **ne** faut **pas** d'instruction **i = i + 1** dans la boucle, **i** change **automatiquement** à chaque tour de boucle.

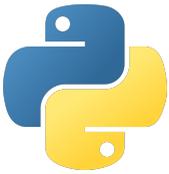
## AU SUJET DE « RANGE »

La fonction **range** est assez générale, la syntaxe complète est **range(debut, fin, pas)**

Instruction	Valeurs successives de <b>i</b>
<code>for i in range(0, 4)</code>	0 ; 1 ; 2 ; 3
<code>for i in range(-6, 6, 2)</code>	-6 ; -4 ; -2 ; 0 ; 2 ; 4
<code>for i in range(10, 5, -1)</code>	10 ; 9 ; 8 ; 7 ; 6

### Remarque :

- Les variables **début**, **fin** et **pas** sont obligatoirement des entiers.
- Si l'incrément (ou le pas) vaut 1, on n'est pas obligé de le spécifier.
- On rencontre également une instruction du type **for i in range(5)**. Dans ce cas, le début vaut forcément 0, on ne précise que la *fin* (non atteinte) et le pas vaut 1. Dans le cas présent, *i* vaut successivement 0, 1, 2, 3, 4.



## CE QU'EST UNE LISTE

En Python, on peut définir une liste comme une collection ordonnée d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets. Les différents éléments ne sont pas forcément de même type.

Voilà un exemple simple d'utilisation d'une liste, dans une console interactive.

```
1 >>> ma_liste = [4,3,8]
2
3 >>> ma_liste[2]
4 8
5
6 >>> ma_liste[0] = "Salut"
7
8 >>> ma_liste
9 ["Salut", 3, 8]
```

Chaque élément est repéré par son **indice** (ou index, ou rang), mais attention, le premier élément est celui d'indice 0, le deuxième d'indice 1 etc...



L'indice commence à 0 !

## QUELQUES FONCTIONS UTILES

Voici les fonctions de base de manipulation de listes :

<code>ma_liste = []</code>	crée une liste vide nommée « ma_liste »
<code>ma_liste.append(9)</code>	ajoute l'élément « 9 » en queue de liste
<code>len(ma_liste)</code>	renvoie le nombre d'éléments (la longueur) de la liste
<code>ma_liste[i]</code>	renvoie l'élément d'indice <i>i</i> contenue dans ma_liste.
<code>del(ma_liste[i])</code>	supprime l'élément d'indice <i>i</i> de la liste.
<code>ma_liste.sort()</code>	trie la liste par ordre croissant
<code>ma_liste[-1]</code>	renvoie le <b>dernier</b> élément de la liste

## CRÉER UNE LISTE PAS À PAS

---

Voici comment on peut créer une liste qui contient tous les multiples de 3 inférieurs à 80

```
1 liste_mul_trois = [] # On débute avec une liste vide.
2 n = 0
3 # On ajoute les multiples de 3 un à un :
4 while n <= 80 :
5     liste_mul_trois.append(n)
6     n = n + 3
```

**Remarque :** Une boucle for est également bien adapté si l'on connaît le nombre d'éléments de la liste à ajouter.

## PARCOURIR UNE LISTE

---

On peut parcourir tous les éléments d'une liste :

par l'intermédiaire des indices des éléments dans une boucle **for** qui boucle pour **i** de 0 à **len(liste)**.

```
liste = [1,2,3,4]
somme = 0

for i in range(len(liste)) :
    somme = somme + liste[i]
```

en accédant directement aux éléments de la liste par l'instruction « **for elem in liste** »

```
liste = [1,2,3,4]
somme = 0

for nombre in liste :
    somme = somme + nombre
```

**Remarque :**

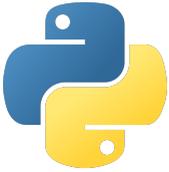
La première solution est la seule valable si l'on souhaite modifier les éléments de la liste durant son parcours.

## UN EXEMPLE TRÈS CLASSIQUE

---

Voici une fonction qui calcule la moyenne des éléments d'une liste.

```
1 def moyenne(liste) :
2     somme = 0
3     n = len(liste)
4     for i in range(n) :
5         somme = somme + liste[i]
6     return somme/n
```



## PRESQUE COMME UNE LISTE...

En python, une chaîne de caractère se manipule comme une liste. Voici un extrait de la console interactive :

```
1 >>> mon_prenom = "Alban"
2 >>> len(mon_prenom)
3 5
4 >>> mon_prenom[0]
5 'A'
6 >>> for lettre in prenom :
7 ...     print(lettre)
8
9 A
10 l
11 b
12 a
13 n
14 >>> mon_prenom[2] = "B" # Oups !
15 Traceback (most recent call last):
16 File "<console>", line 1, in <module>
17 TypeError: 'str' object does not support item assignment
```



Contrairement aux listes, dont on peut modifier les éléments, on ne peut pas modifier une lettre d'une chaîne de caractère. L'instruction `mon_prenom[2] = "B"` est invalide.

Pour modifier une chaîne, il faut en créer une nouvelle qui contiendra la version modifiée.

```
1 def NomCool(nom) :
2     new_chaine = "" # Une chaîne vide
3     for lettre in nom :
4         if lettre in ["a","A"] :
5             new_chaine = new_chaine + "@"
6         else :
7             new_chaine = new_chaine + lettre
8     return new_chaine
9 print(NomCool("Alban"))
10 # On obtient :
11 >>>> @lb@n
```

## OPÉRATIONS SUR LES CHAÎNE DES CARACTÈRES

Opérateur	Exemple	Commentaire
	<code>ch = ""</code>	Crée une chaîne de caractères vide
<code>+</code>	<code>"Bon" + "jour" → "bonjour"</code>	Concaténation
<code>ord()</code>	<code>ord("A") → 65</code>	Code Ascii du caractère
<code>chr()</code>	<code>chr(66) → "B"</code>	Caractère associée au code Ascii

Code	Car.	Code	Car.	Code	Car.	Code	Car.	Code	Car.	Code	Car.
32	space	48	0	64	@	80	P	96	'	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

On donne ici les codes ASCII des caractères affichables. Les caractères de code 1 à 31 ne sont pas affichables, ce sont **des caractères de contrôle**.

## CARACTÈRES DE CONTRÔLE

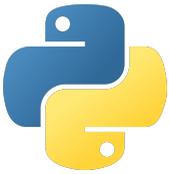
Un retour chariot **est un caractère**, mais ne s'affiche pas (il correspond à la touche "entrée" du clavier). Il génère simplement un retour à la ligne lors de l'affichage de la chaîne. Les deux caractères non affichables les plus utiles sont les suivants :

nom	code python	code ASCII	fonction
retour chariot	<code>"\n"</code>	13	retour à la ligne
tabulation	<code>"\t"</code>	9	décalage à la colonne suivante

```

1 >>> print("bonjour\nle monde")
2 bonjour
3 le monde

```



---

## LE CHOIX DES NOMS

---

Pour la lisibilité d'un programme, il est très important de choisir des noms de fonctions et de variables **parlants**. De plus, il est bon de garder une certaine cohérence quant à la façon de choisir les noms des variables et fonctions.

On peut trouver les conseils « officiels » dans le **PEP8**<sup>1</sup>.

L'exemple suivant ne suit pas directement ce document. En effet, il est conseillé de réserver le **CamelCase** pour le nom des classes, mais comme nous ne les utilisons pas au lycée, nous pouvons nous en servir pour les fonctions. Par exemple, on peut choisir les conventions

- `lower_case_with_underscores` pour les noms de variables.
- `CapWord`, aussi nommé **CamelCase**, pour les noms de fonctions.

On peut également préfixer le nom des variables par une information sur leur type.

- `liste_notes`
- `indice_du_maxi`
- `nb_elements`

Des noms parlants et cohérents participent grandement à la clarté d'un programme.

---

## L'AUTO-DOCUMENTATION DE PYTHON

---

Dans la console interactive, on peut obtenir de l'aide sur les fonctions des bibliothèques chargée en mémoire à l'aide de ?.

```
1 >>> import random as rd
2
3 >>> rd.randint?
4 Return random integer in range [a, b], including both end
   points.
```

On peut parfois avoir plus de détails avec ??

Essayer par exemple, dans la console, `import math` puis `math??`

---

1. <https://www.python.org/dev/peps/pep-0008/>

## LES COMMENTAIRES

---

Il y a deux façons d'introduire des commentaires dans un programme python

1. À l'aide du caractère # pour les commentaires sur une seule ligne.
2. En encadrant un bloc de texte par trois guillemets """

```
1 def Maximum(liste) :
2     """
3     Je suis un commentaire multilignes.
4     Il est d'usage de proposer une courte explication
5     de la fonction à cet emplacement. (docstring)
6     """
7     maxi = liste[0]
8     for element in liste :
9         if element > maxi :
10             maxi = element
11     return maxi # commentaire sur une ligne.
```

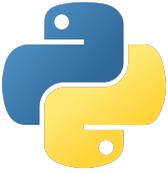
**Remarque :** Le `docstring`, c'est à dire le commentaire multi-lignes qui l'on place juste après la déclaration de la fonction, est utilisé par Python à des fin d'auto-documentation. Il est affichée dans la console interactive lors de l'instruction `MaFonction?`.

## BLOC D'INSTRUCTION

---

En python, un **bloc d'instruction** est une suite d'instructions **ayant toutes le même décalage** par rapport à la marge gauche du document. On dit que les instructions ont la même **indentation**.

```
1 import turtle as t
2 n = 6
3 for i in range(n) :
4     t.forward(10)
5     t.left(360/n)
6 print("figure terminée")
```



# TRACER UNE COURBE

Fiche synthèse n° 11

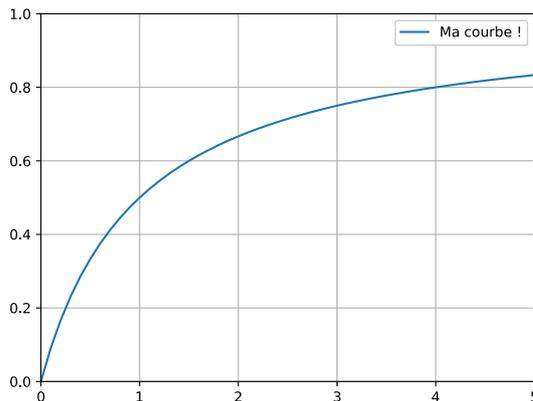
Pour tracer des courbes, nous utilisons la bibliothèque `matplotlib`<sup>1</sup>

Dans la version minimale, il suffit de remplir deux listes de valeurs (de même taille) : une d'abscisses, et une avec les ordonnées correspondantes. On appelle ensuite la fonction `plot` avec les deux listes en paramètres.

```
1 import matplotlib.pyplot as plt
2 X = []
3 Y = []
4 x = 0
5 while x <=5 :
6     X.append(x)
7     Y.append(x/(1+x))
8     x = x + 0.1
9
10 plt.plot(X, Y)
11 plt.show()      # Ne pas oublier d'afficher la courbe !
```

On peut améliorer l'affichage en remplaçant, par exemple, les lignes 11 et 12 par

```
1 plt.plot(X, Y, label = "Ma courbe !")
2 plt.xlim(0,5)      # Limites du tracé en x
3 plt.ylim(0,1)     # Limites du tracé en y
4 plt.grid()        # On ajoute une grille
5 plt.legend()      # On ajoute la légende
6 plt.show()        # Et on affiche le tout !
```



1. On pourra se reporter à la page : <http://matplotlib.org> puis aux onglets « exemples » ou « docs » pour exploiter au mieux cette bibliothèque.