

# Python

Initiation pour un usage en Sciences





# Table des matières

<b>1</b>	<b>Débuter avec Python</b>	<b>5</b>
<b>1.1</b>	<b>Python comme machine à calculer</b>	<b>5</b>
1.1.1	Environnement de travail	5
1.1.2	Opérations de bases	5
1.1.3	Opérations plus complexes	6
<b>1.2</b>	<b>Fonctions</b>	<b>7</b>
1.2.1	Un premier exemple	7
1.2.2	Notion de paramètre	8
1.2.3	Plusieurs paramètres	9
<b>2</b>	<b>Variables</b>	<b>11</b>
<b>2.1</b>	<b>Notion de variable</b>	<b>11</b>
<b>2.2</b>	<b>Affectations</b>	<b>11</b>
2.2.1	Mise à jour d'une variable	13
2.2.2	Différence Variable – Fonction	13
2.2.3	Affectations multiples	14
<b>2.3</b>	<b>Conversions</b>	<b>15</b>
<b>2.4</b>	<b>Scripts</b>	<b>15</b>
2.4.1	Scripts dans IDLE	15
2.4.2	Interaction	16
2.4.3	Commentaires	16
<b>3</b>	<b>Structures conditionnelles</b>	<b>19</b>
<b>3.1</b>	<b>Booléens</b>	<b>19</b>
3.1.1	Opérateurs booléens	20
<b>3.2</b>	<b>Déroulement d'un programme</b>	<b>20</b>
3.2.1	Séquence d'instructions	20
3.2.2	Exécution conditionnelle	20
3.2.3	Répétitions	24
<b>4</b>	<b>Listes</b>	<b>31</b>
<b>4.1</b>	<b>Définition</b>	<b>31</b>
<b>4.2</b>	<b>Utilisation</b>	<b>31</b>
4.2.1	Accès et affichage	31
4.2.2	Mutation	32

4.3	Parcourir une liste	34
4.4	Liste en compréhension	34
4.5	Exercices	35
<b>5</b>	<b>Chaînes de caractères</b>	<b>37</b>
5.1	Qu'est-ce qu'une chaîne de caractères ?	37
5.1.1	Encodage	37
5.1.2	Le type <code>str</code>	37
5.2	Manipulations	39
5.2.1	Sous-chaînes	39
5.2.2	Concaténation	39
5.2.3	Répétition	39
5.2.4	Comparaison	40
5.2.5	Chaînes en tant qu'objets	40
5.3	Formater une chaîne	41
5.4	Exercices	43
<b>A</b>	<b>Nombres complexes</b>	<b>45</b>
A.1	Implémentation native	45
A.1.1	Définition	45
A.1.2	Parties réelles et imaginaires	45
A.1.3	Opérations	46
A.2	Librairie <code>cmath</code>	46
A.2.1	Forme trigonométrique	47
A.2.2	Fonctions spécifiques	47
	<b>Index</b>	<b>49</b>

# 1 | Débuter avec Python

## 1.1 Python comme machine à calculer

### 1.1.1 Environnement de travail

Dans le menu Démarrer, taper « IDLE » et choisir IDLE Python dans la liste des programmes proposés. Vous venez d'ouvrir une console d'interprétation du langage Python. On va pouvoir parler à l'ordinateur en utilisant ce langage et lui demander d'exécuter certaines tâches. On saisi une phrase puis on tape sur entrée et l'interpréteur traduit la ligne de commandes et fait le travail.

Commençons par des calculs.

### 1.1.2 Opérations de bases

Il existe plusieurs opérateurs pour faire les opérations courantes

```
1 >>> 1 + 1          # Addition
2 2
3 >>> 9 - 13         # Soustraction
4 -4
5 >>> 2 * 3          # Multiplication
6 6
7 >>> 17 / 4         # Division
8 4.25
9 >>> 2 ** 4          # Puissance
10 16
11 >>> (2 - 3) * 6 + 4 / 2 # Memes priorités qu'en maths
12 -4
```

Cas de la division entière (ou encore la division euclidienne).

```
13 >>> 17 / 4        # Division décimale
14 4.25
15 >>> 17 // 4        # Division entière _explicite_
16 4
17 >>> 17,0 / 4       # ATTENTION : La virgule en Python est un point (notation,
18 (17, 0)            # américaine) d'où le résultat inattendu
```

On a une syntaxe pratique pour l'écriture scientifique et plus généralement pour les multiplications par des puissances de 10 :

```
19 >>> 2e3            # 2 x 10^3
20 2000.0
21 >>> 2E3            # 'E' majuscule marche aussi
22 2000.0
23 >>> 3e-3           # 3 x 10^-3
24 0.003
25 >>> 31.27e5        # 31,27 x 10^5
26 3127000.0
```

**Exercice 1.1** Calculer les valeurs des expressions suivantes avec Python :

$$A = (4 \times 5 - 10) \div 3$$

$$B = \frac{4}{3} + 5 \times 8 - 3$$

$$C = (3^2 - 1)(3 - 5^2)$$

$$D = 2 \times (-3)^2 - 4 \times (-3) + 7$$

$$E = (4^6 - 12^3) \times (1 - 9^4)$$

$$F = \frac{4 \times 10^{21} + 12 \times 10^{23}}{8 \times 10^{25}}$$

**Exercice 1.2** Sachant que la vitesse de la lumière est de  $299\,792\,458 \text{ m.s}^{-1}$ , donner le nombre de kilomètres parcouru par la lumière en un an (soit une année lumière).

**Exercice 1.3** Utiliser Python pour calculer le reste de la division euclidienne de  $2^{32} + 1$  par 641. On pourra commencer par se débrouiller avec la division entière (" $//$ "), puis on comparera avec l'opération "%" existant déjà en Python.

**Exercice 1.4** Soit une fonction

$$h : x \mapsto 3x^2 - 5x + 5$$

Calculer à l'aide de Python les images de  $-2$ ,  $1$  et  $7$ .

### 1.1.3 Opérations plus complexes

Pour calculer des opérations plus complexes comme des racines carrées par exemple, on doit dire à Python de regarder dans un livre car il ne sait pas le faire tout seul. On doit faire appel à une **librairie** qui va apporter plus de fonctionnalité au langage.

Pour faire des calculs en mathématiques, on fait appel à la librairie... **math** ! On utilise pour cela les mots clés **from** et **import**.

```
27 >>> from math import sqrt      # 'sqrt' pour 'square root', la racine carrée
28 >>> sqrt(2)
29 1.4142135623730951
```

Dans l'exemple ci-dessus, on indique à Python qu'on souhaite utiliser la fonction **sqrt()** pour calculer une racine carrée. Une fois cette ligne saisie, Python connaîtra cette fonction.

On peut aussi indiquer à Python d'importer *toute* une librairie.

Il existe pour cela deux méthodes. Dans la première, on charge *seulement* la librairie dans la mémoire de Python. Il faudra alors *référencer* les fonctions à la librairie importée. Par exemple, on dira **math.sqrt()** et non plus **sqrt()**.

```
30 >>> import math                # on importe toute la librairie 'math'
31 >>> math.sqrt(2)               # on référence la fonction 'sqrt'
32 1.4142135623730951
33 >>> math.cos(2*math.pi)       # Python mesure les angles en radians !
34 1.0
35 >>> math.degrees(math.pi/4)   # une conversion en degrés d'un angle en radian
36 45.0
```



Attention : bien faire la différence entre une *fonction* comme **math.cos()** qui nécessite des parenthèses (on passe un nombre comme paramètre) et une *constante* comme **math.pi** qui ne nécessite pas de parenthèses.

Dans la deuxième technique d'utilisation d'une librairie, on dit à Python de charger en mémoire toutes les fonctions et constantes de la librairie en une fois. On se sert alors du caractère `*` (appelé *wild card*). Dans ce cas, on a plus à référencer la fonction à la librairie.

Cette dernière méthode d'importation d'une librairie est toutefois déconseillée : en effet, si on utilise plusieurs librairies et que deux fonctions portent le même nom, Python utilisera comme librairie la dernière importée seulement.

```
37 >>> from math import *
38 >>> cos(pi)
39 -1.0
```

**Exercice 1.5** Calculer les expressions suivantes à l'aide de Python :

$$G = \cos\left(\frac{3\pi}{2}\right)$$

$$H = \tan(45^\circ)$$

$$I = \sqrt{2 \times 5^2 - 9 \times 5 + 9}$$

$$J = \sqrt{5 \cos\left(\frac{\pi}{5}\right) - 3 \sin\left(-\frac{\pi}{5}\right)}$$

**Exercice 1.6** Parce que ces bases de numération ont joué un rôle dans l'histoire de l'informatique, Python dispose (dans sa version de base) de fonctions de conversion en binaire (base 2), en octale (base 8) ou en hexadécimal (base 16). Utiliser les fonctions "bin", "oct" et "hex" pour convertir les entiers 17 et 65 en base 2, 8 ou 16. Que penser du format d'affichage des résultats ?

## 1.2 Fonctions

Il est possible dans Python de définir ses propres *fonctions*. C'est un bout de code, quelques lignes de commandes qui décrivent une série de tâches, que Python va mettre en mémoire et que nous allons pouvoir réutiliser tant que la session est ouverte.

### 1.2.1 Un premier exemple

Pour définir une fonction on utilise le mot clé `def`, suivi du nom de la fonction<sup>1</sup> suivi d'un couple de parenthèses, suivi de deux points `:`<sup>2</sup>.

Ensuite, il faut faire comprendre à Python quelles sont les lignes qui font partie de la fonction. Pour cela, on utilise une indentation, c'est à dire un écart de 4 espaces de la marge.

```
40 >>> def resultat() :           # 'signature' de la fonction
41 ...     return 4**2 - 3        # bloc de code (notez l'indentation)
42 ...
43 >>> resultat()                # appel de la fonction
44 13
```

Ci-dessus, aux deux premières lignes, nous avons écrit une fonction `resultat()` qui retourne (utilisation de la commande `return`) le résultat d'un calcul :  $4^2 - 3$ . Pour appeler la fonction, on saisit juste son nom suivi des parenthèses. Attention à ne pas oublier les parenthèses par contre, sinon le résultat est inattendu !

```
45 >>> resultat
46 <function resultat at 0x7f1a59c4aea0>
```

On peut utiliser cette fonction dans un autre calcul :

1. Attention, il ne faut pas utiliser un mot déjà employé !
2. Cette première ligne est la *signature* de la fonction `resultat()`

```

47 >>> 2 * resultat()
48 26

```

Par contre, notre fonction n'est pas très intéressante car elle fait toujours le même calcul. Comment pourrait-on faire pour qu'elle calcule  $x^2 - 3$  pour une valeur de  $x$  que nous aurions choisi ? C'est ce que nous allons voir dans un deuxième exemple.

### 1.2.2 Notion de paramètre

Lorsque nous avons utilisé la fonction `math.sqrt()` pour calculer  $\sqrt{2}$ , nous avons passé le nombre 2 comme *argument* à la fonction :

```

49 >>> math.sqrt(2)
    ↪ # on donne '2' comme argument à la fonction 'sqrt()'
50 1.4142135623730951

```

On va donc utiliser le même principe dans la fonction qui doit calculer  $x^2 - 3$ .

```

51 >>> def f(x) :                # définition du paramètre 'x' avec la fonction
52 ...     return x**2 - 3        # utilisation du paramètre 'x' dans le calcul
53 ...

```

Entre les parenthèses dans la signature de la fonction, on introduit un *paramètre* nommé ici `x` (mais nous aurions pu lui donner un tout autre nom comme `choucroute` mais c'est plus long). Cela indique à Python que lorsque nous allons appeler la fonction, il faudra lui passer un argument, en l'occurrence un nombre.

```

54 >>> f(2)                      # appel de la fonction avec l'argument '2'
55 1
56 >>> f(4)                      # appel de la fonction avec l'argument '4'
57 13

```

Par exemple, lorsque nous appelons la fonction avec l'argument 2, Python associe le nombre 2 au paramètre `x`. Lorsqu'on réutilise le paramètre dans l'expression du calcul, Python remplace  $x$  par 2.

Si nous appelons la fonction `f` sans argument (comme la fonction `resultat()`), on a un message d'erreur : Python nous dit qu'il ne sait pas quoi faire parce qu'il attend un argument !

```

58 >>> f()
59 Traceback (most recent call last) :
60   File "<stdin>", line 1, in <module>
61   TypeError : f() missing 1 required positional argument : 'x'

```

**Exercice 1.7** Écrire une fonction qui calcule l'expression  $(x - 3)(2x + 9)$  pour une valeur de  $x$  choisie par l'utilisateur. ■

**Exercice 1.8** Reprendre l'exercice 1.4 mais le traiter à l'aide d'une fonction. ■

**Exercice 1.9** Écrire les fonctions qui calculent les grandeurs suivantes :

1. l'aire d'un carré connaissant la longueur de son côté ;
2. le périmètre d'un cercle connaissant son rayon (utiliser la constante Python `math.pi`). ■



**Exercice 1.10** Appel d'une fonction depuis une autre fonction.

1. Écrire une fonction `cube()` qui renvoie le cube de son argument.
2. Écrire une fonction `volume_sphere()` qui renvoie le volume d'une sphère de rayon  $r$  passé en argument. Cette fonction devra utiliser la fonction `cube()` précédente ainsi que la constante `math.pi`.

### 1.2.3 Plusieurs paramètres

On peut utiliser le même concept avec plusieurs paramètres. Il suffit de séparer par une virgule les paramètres dans la signature et les arguments dans l'appel de la fonction :

```
62 >>> def hypotenuse(a, b) :
63 ...     return math.sqrt(a**2 + b**2)
64 ...
65 >>> hypotenuse(3, 4)
66 5.0
```

La fonction `hypotenuse()` utilise deux arguments `a` et `b`. Au fait, que fait cette fonction ?

**Exercice 1.11** Écrire une fonction `aire()` qui prend en paramètres la longueur et la largeur d'un rectangle et qui renvoie son aire.

**Exercice 1.12** Écrire une fonction `volume_boite()` qui calcule le volume d'une boîte connaissant la largeur `l`, la hauteur `h` et la profondeur `p` qui seront les paramètres de la fonction.

**Exercice 1.13 — Conversion de degrés.** Si  $F$  est une température en degré Fahrenheit, la température en degré Celsius est donnée par  $\frac{5}{9}(F - 32)$ .

Écrire une fonction `convert_F_to_C()` qui convertit une température donnée en degrés Fahrenheit en degrés Celsius.

**Exercice 1.14 — Triangle.** 1. Écrire une fonction `perimetre_triangle()` qui prenne en paramètres la longueur de ses trois côtés et qui renvoie la valeur de son périmètre.

2. Écrire une fonction `aire_triangle()` qui prenne en paramètres la longueur de ses trois côtés et qui renvoie la valeur de son aire.

On donne la formule suivante<sup>a</sup> qui permet de calculer l'aire  $\mathcal{A}$  connaissant les longueurs  $a$ ,  $b$  et  $c$  des trois côtés :

$$\mathcal{A} = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{avec} \quad p = \frac{1}{2}(a+b+c)$$

<sup>a</sup>. que l'on doit à Héron d'Alexandrie, un mathématicien grec du I<sup>er</sup> siècle après J-C.



## 2 | Variables

Dans certains cas, on peut avoir besoin d'un nombre plusieurs fois dans un programme. Plutôt que d'avoir à le retaper à chaque fois son expression, Python nous propose un moyen simple de manipuler les nombres : les *variables*.

### 2.1 Notion de variable

1. Une variable est basiquement une boîte avec une étiquette (voir figure 2.1a). Côté utilisateur : c'est l'étiquette, c'est-à-dire le nom de la variable. Côté ordinateur : c'est la boîte, en fait une adresse mémoire. On peut faire appel à la fonction `id()` pour voir cette adresse mémoire.
2. Différent *types* : entier, réel, chaîne de caractères, liste, etc. La fonction `type()` renseigne sur le type d'une variable.
3. Typage automatique : en fonction de ce que l'on met dans la boîte, Python sait ce qu'il a le droit de faire avec.
4. Il faut respecter un certain nombre de règles pour nommer les variables :
  - (a) caractères ou chiffre ou tiret-bas (`_`) seulement,
  - (b) le premier caractère est obligatoirement une lettre,
  - (c) sensible à la casse (`chaîne` et `Chaîne` sont deux variables différentes)
  - (d) certains mots sont dits *réservés* : ils désignent des commandes Python et ne pas être utilisés comme nom de variables. On peut les connaître depuis l'interpréteur directement :

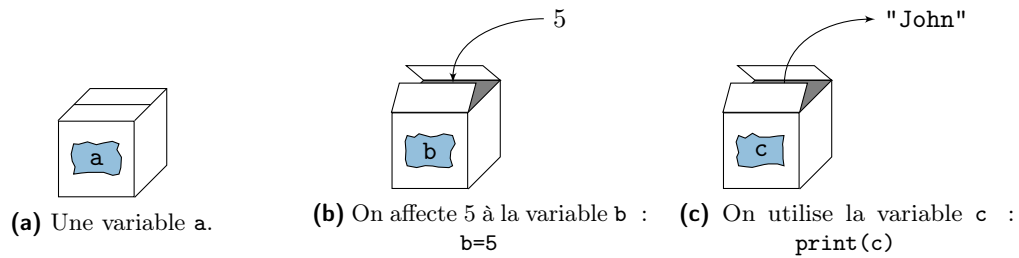
```
67 >>> import keyword
68 >>> keyword.kwlist
69 ['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
   ↪ 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
   ↪ 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
   ↪ 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
   ↪ 'with', 'yield']
```

### 2.2 Affectations

Avec Python, on déclare la variable en même temps qu'on lui attribue une valeur. En reprenant l'analogie des boîtes, il n'existe pas de boîte vide : cela n'a pas de sens de préparer des boîtes à l'avance avec Python, on les crée au moment où on en a besoin.

Pour *affecter* (ou assigner) une valeur à une variable (c'est-à-dire mettre une étiquette à une boîte **et** y mettre une certaine valeur, ces deux opérations se faisant en même temps, voir figure 2.1b), on utilise l'opérateur `=`.

Figure 2.1 – Une variable est une boîte étiquetée.



```

70 >>> a = 4           # Affecter la valeur 4 à la variable 'a'
71 >>> id(a)          # Adresse mémoire de la variable 'a'
72 140443874558720
73 >>> type(a)        # Afficher le type de 'a'
74 <type 'int'>
75 >>> a              # Afficher la valeur de 'a' (ne marche pas dans un script)
76 4
77 >>> b = 5          # Affecter la valeur 5 à la variable 'b'

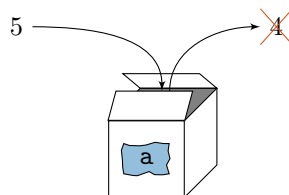
```

Si on utilise le même nom de variable plusieurs fois, c'est la dernière valeur qui est retenue, Python ayant jeté l'ancienne et mis l'autre à la place (voir figure 2.2) :

```

78 >>> a=4
79 >>> print(a)
80 4
81 >>> a=5
82 >>> print(a)
83 5

```

Figure 2.2 – La variable **a** contenait 4 et on y stocke 5 : le nombre 4 est jeté

On peut utiliser les variables dans des calculs voir figure 2.3 : Python les remplace par leur valeur. S'il trouve une boîte marquée **a**, il l'ouvre et utilise ce qu'il trouve dedans à la place de **a** dans l'expression.

```

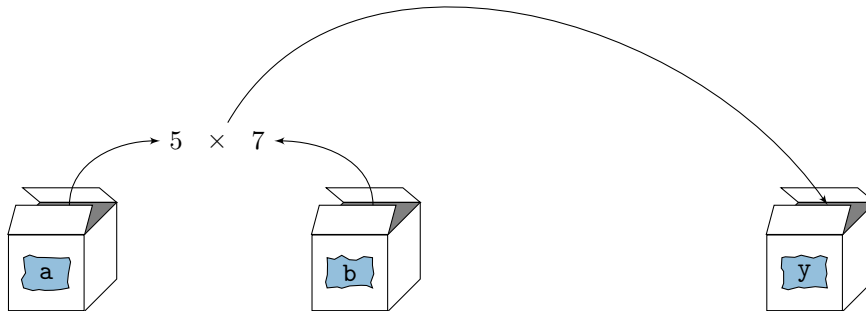
84 >>> a * b           # Calcul avec les variables 'a' et 'b'
85 20

```

**Exercice 2.1** Stocker la valeur du rayon d'un cercle dans une variable **r**.  
Utiliser cette variable **r** pour calculer la longueur du cercle et l'aire du disque correspondant. ■

**R** Pour afficher la valeur d'une variable, on peut se contenter de taper son nom dans l'interpréteur. Toutefois, c'est une fonctionnalité de l'interpréteur qu'on ne retrouve pas dans un programme. Il faut alors utiliser la fonction `print()`.

**Figure 2.3** – Utilisation des variables **a** et **b** dans une opération puis affectation du résultat dans une variable **y**.



```
86 >>> print(a)           # Afficher la valeur de la variable 'a'
87 4
```

On peut passer autant d'arguments qu'on souhaite à la fonction `print()`, on les sépare par une virgule.

```
88 >>> print("Score :", a) # Utilisation de 'a' dans une phrase.
89 Score : 4
```

**Exercice 2.2** Stocker "John" dans la variable `name` et 23 dans la variable `age`. À l'aide de ces deux variables et de la fonction `print()`, afficher la phrase "John a 23 ans". ■

### 2.2.1 Mise à jour d'une variable

Dans une affectation, Python évalue en premier le membre à droite du signe `=` et met la valeur de ce membre dans la variable. Ainsi, on peut utiliser la variable elle-même dans le membre de droite.

Par exemple, pour augmenter la variable `a` de 1, on fera :

```
90 >>> a=5                # on affecte 5 à a
91 >>> print(a)
92 5
93 >>> a=a+1              # on augmente a de 1
94 >>> print(a)
95 6
96 >>> a+=1               # raccourci...
97 >>> print(a)
98 7
```

**Exercice 2.3** Soit une variable `n`. On cherche à mettre à jour sa valeur. Montrer comment on peut :

1. Augmenter `n` de 5.
2. Tripler `n`.
3. Diviser `n` par 10.

Donner deux écritures différentes pour chaque question. ■

### 2.2.2 Différence Variable – Fonction


On peut définir une *expression* à partir de variables, toutefois si une des variables de l'expression change de valeur, l'expression n'est pas mise à jour. Il faudra utiliser le concept de *fonction* (voir section ?? page ??).

```

99 >>> y = 3 * a - 2 * b      # Affecter une expression à la variable 'y'
100 >>> print(y)
101 2
102 >>> a = 2                  # Nouvelle valeur pour la variable 'a'
103 >>> print(y)               # La variable 'y' n'est pas mis a jour
104 2

```

Ci-dessus, on définit `y` en fonction de deux autres variables `a` et `b`. Il y aura dans `y` la valeur de l'expression au moment où elle est définie. Si on change les valeurs de `a` ou `b` après, la valeur de `y` n'est pas affectée.

 Attention à ça :

```

105 >>> a = 6
106 >>> b = a
107 >>> print(b)
108 6
109 >>> a = 7
110 >>> print(b)               # La variable 'b' n'est pas liée a 'a'
111 6

```

### 2.2.3 Affectations multiples

Python autorise les multiples affectations, elles sont parfois une manière élégante d'aborder un problème. Si elles sont possibles, il faut les manipuler avec précaution car elles sont parfois difficile à interpréter.

```

112 >>> a, b = 7, 9           # Affecte 7 a la variable 'a' et 9 a 'b'
113 >>> print(a)
114 7
115 >>> print(b)
116 9
117 >>> x = y = 3              # Affecte 3 aux variables 'x' et 'y'
118 >>> print(x)
119 3
120 >>> print(y)
121 3

```

**Exercice 2.4** Dans chacun des cas suivants, prévoir ce qui est affiché en sortie puis vérifier la réponse dans un interpréteur.

#### Code

```

1 >>> a = 11
2 >>> b = a - 7 ; a = a + 3
3 >>> a, b

```

#### Code

```

1 >>> a = 11
2 >>> a, b = a + 3, a - 7
3 >>> a, b

```

#### Code

```

1 >>> a = 11
2 >>> a += 3 ; b = a - 7
3 >>> a, b

```

#### Code

```

1 >>> a = 11
2 >>> b, a = a - 7, a + 3
3 >>> a, b

```

**Exercice 2.5** Soit `a` et `b` deux variables ayant respectivement pour valeurs 10 et 3. Montrer deux moyens d'échanger les valeurs de `a` et `b`. ■

## 2.3 Conversions

Quelquefois, on a besoin de convertir une variable d'un certain type dans un autre type, pour pouvoir faire certaines opérations interdites avec un type mais autorisées avec un autre. On utilise le nom du type comme une fonction.

```

4 >>> a = "3"
5 >>> 2 + a          # Python ne sait pas faire ! Et il le dit !!!
6 Traceback (most recent call last) :
7   File "<stdin>", line 1, in <module>
8   TypeError : unsupported operand type(s) for + : 'int' and 'str'
9 >>> 2 + int(a)
10 5

```

Si on essaie d'additionner un entier avec une chaîne de caractères, on obtient un message d'erreur car Python ne sait pas faire. Il faut donc convertir la variable `a` qui est du type chaîne de caractères en entier en utilisant la fonction `int()`.

Quelques fonctions de conversion utiles : `float()` pour convertir un objet en décimal, `str()` pour convertir un objet en chaîne de caractères, `hex()` et `oct()` pour convertir un objet en nombre en base 16 ou 8 respectivement (l'objet produit est une chaîne), etc.

**Exercice 2.6** Tenter de convertir en entier les chaînes de caractères suivantes : "78", "7.8", "7\*8", "nombre". Interpréter les résultats. ■

## 2.4 Scripts

En dehors de l'interpréteur qui permet d'avoir le résultat d'une ligne de commande directement, on peut consigner plusieurs lignes de commandes Python dans un même fichier : un *script*.

Un *script* est de manière plus générale un fichier dans lequel on a inscrit une liste d'instructions dans un certain langage compréhensible par l'ordinateur. Plus spécifiquement, les scripts écrit en Python utilisent l'extension `*.py` comme nom de fichier, comme par exemple dans `test.py`.

### 2.4.1 Scripts dans IDLE

1. Pour *éditer* un script dans IDLE, on clique sur Fichier > Nouveau ou on tape `[Ctrl]+[N]`. Une nouvelle fenêtre apparaît alors. On remarque l'absence de prompt (pas de chevrons `>>>`) qui indique que nous ne sommes plus dans l'interpréteur mais dans un éditeur. On saisit le texte et on tape `[Enter]` à la fin de chaque ligne.

En règle générale, on fait commencer un script Python par cette ligne :

```

11 # -*- coding : utf8 -*-

```

Cette ligne indique à la machine quel est l'encodage utilisé pour écrire le script, c'est-à-dire le type de représentation numérique de chaque caractère <sup>1</sup>.

Même si elle est utile, cette ligne est optionnelle et n'est pas nécessaire à la bonne exécution du script.

2. Pour *sauvegarder* un script, on choisit Fichier > Sauvegarder ou on tape `[Ctrl]+[S]`. On choisit alors le dossier de destination <sup>2</sup>.

1. Pour plus d'informations sur le codage des caractères, voir [http://fr.wikipedia.org/wiki/Codage\\_de\\_caract%C3%A8res](http://fr.wikipedia.org/wiki/Codage_de_caract%C3%A8res).

2. Penser à sauvegarder dans votre espace personnel pour retrouver votre script sur tout le réseau !

3. Pour *exécuter* un script python depuis IDLE, on tape la touche **F5** ou on choisit Run > Run Module et l'ordinateur exécute les instructions inscrites dans le script les unes après les autres, dans l'ordre où elles ont été écrites. Le résultat s'affiche dans une nouvelle fenêtre.

### 2.4.2 Interaction

On peut depuis un programme demander à l'utilisateur saisir une séquence de caractères au clavier et stocker la chaîne ainsi produite dans une variable. C'est la fonction `input()`.

Code	Résultat
<pre>1 a=input("nombre ? ") 2 print(a)</pre>	<pre>nombre ? 5 5</pre>

**Exercice 2.7** Demander à l'utilisateur son nom et stocker le résultat dans `name`. Afficher ensuite "Bonjour <name> !" où <name> sera remplacé par le contenu de la variable `name` (par exemple si `name` contient "John", on affichera "Bonjour John !"). ■

**R** La fonction `input()` renvoie une chaîne de caractères. Il faudra convertir au besoin pour faire un calcul.

Code	
<pre>1 a=input("nombre ? ") 2 print("Double :", 2*a) 3 print("Vrai double :", 2*int(a))</pre>	<pre># pas de conversion # conversion de 'a' en entier</pre>
Résultat	
<pre>nombre ? 5 Double : 55 Vrai double : 10</pre>	

**Exercice 2.8** Écrire un programme qui demande à l'utilisateur le rayon d'une sphère puis affiche son volume. On pourra reprendre la fonction écrite à l'exercice 1.10. ■

### 2.4.3 Commentaires

Toute ligne d'un script ou toute fin de ligne qui commence par le caractère `#` est ignoré par Python. On appelle cette ligne un *commentaire* car il sert à la personne qui écrit le script à expliquer ce qu'elle a voulu faire.

Lorsque le commentaire comprend plusieurs lignes, on peut le mettre entre deux triples double guillemets :

<pre>4 """ 5 Mon long commentaire sur 6 plusieurs lignes ici 7 """ 8 9 # Cette ligne est totalement ignorée par Python 10 11 print "Hello world !" # après le dièse, c'est ignoré</pre>	
---	--



Il faut prendre l'habitude de commenter vos programmes. On pourra prendre le code qui suit en exemple :

```

12 def hypotenuse(cote1, cote2) :
13     """
14     Calcule la longueur d'une hypoténuse connaissant
15     les côtés de l'angle droit (passés en paramètres).
16     """
17     return math.sqrt((cote1)**2 + (cote2)**2)
18
19 #----- Calcul de AC -----
20 AB = 9      # Premier côté
21 BC = -3     # deuxième côté
22
23 # Afficher la longueur de longueur AC
24 print hypotenuse(AB, BC)

```

Il faut expliquer ce que stockent les variables, donner des éléments d'explication d'un algorithme, dire ce que fait une fonction, etc. C'est important pour les autres qui lisent votre code mais aussi pour vous-même lorsque vous reprenez un script que vous avez écrit il y a longtemps : on ne se souvient jamais de la manière dont on a traité un problème quand le code n'est pas récent.

**Exercice 2.9 — Jour de la semaine.** Écrire un programme qui demande à l'utilisateur une date (successivement le jour, le mois, l'année) et qui retourne le rang du jour dans la semaine auquel correspond cette date.

En prenant  $m$  pour le mois,  $d$  pour le jour et  $y$  pour l'année (pour janvier,  $m$  doit avoir la valeur 1, pour février,  $m$  doit avoir la valeur 2, etc.), les formules<sup>a</sup> suivantes (convenant pour le calendrier grégorien) donnent au final  $d_0$  où 0 correspond à dimanche, 1 à lundi, etc. :

$$\begin{aligned}
 y_0 &= y - \frac{14 - m}{12} \\
 x &= y_0 + \frac{y_0}{4} - \frac{y_0}{100} + \frac{y_0}{400} \\
 m_0 &= m + 12 \left( \frac{14 - m}{12} \right) - 2 \\
 d_0 &= \left( d + x + \frac{31m_0}{12} \right) \mod 7
 \end{aligned}$$

#### Exemple

Quel est le jour de la semaine correspondant au 14 février 2000 ?

$$\begin{aligned}
 y_0 &= 2000 - 1 = 1999 \\
 x &= 1999 + \frac{1999}{4} - \frac{1999}{100} + \frac{1999}{400} = 2483 \\
 m_0 &= 2 + 12 \times 1 - 2 = 12 \\
 d_0 &= \left( 14 + 2483 + \frac{31 \times 12}{12} \right) \mod 7 = 2500 \mod 7 = 1
 \end{aligned}$$

La réponse est donnée par  $d_0 = 1$  donc le 14 février 2000 était un lundi. ■

<sup>a</sup>. Dans ces formules, les divisions sont des divisions entières et  $\mod$  est l'opération % en Python donnant le reste de la division entière.

**Exercice 2.10 — Distance parcourue par un objet.** On souhaite écrire un programme qui calcule la distance parcourue par un objet dans certaines conditions.

1. La distance parcourue en mètres après  $t$  secondes par un objet lancé en ligne droite à la vitesse  $v_0$  (en mètres par seconde) depuis une position initiale  $x_0$  (en mètres) est donné par l'expression  $x_0 + v_0 t + \frac{gt^2}{2}$  où  $g$  est une constante (accélération de la pesanteur) égale à environ 9,806 65.

Écrire un programme qui demande à l'utilisateur les trois valeurs  $x_0$ ,  $v_0$  et  $t$  et calcule la distance parcourue correspondante.

2. En réalité, l'accélération de la pesanteur  $g$  dépend de la latitude  $L$  (mesurée en radians) et de l'altitude  $h$  (mesurée en mètre). La formule suivante donne une valeur approchée de la valeur de  $g$  en fonction de la latitude et de l'altitude faible en regard du rayon terrestre :

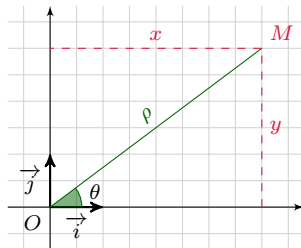
$$g = 9,780327 \times (1 + 5,3024 \times 10^{-3} \times \sin^2(L) - 5,8 \times 10^{-6} \times \sin^2(2 \times L) - 3,086 \times 10^{-7} \times h)$$

Modifier le programme précédent pour qu'il prenne en compte cette formule en demandant la latitude  $L$  en degrés<sup>a</sup> et l'altitude  $h$  à l'utilisateur.

Penser à écrire des fonctions pour les différents calculs et aussi à utiliser les fonctions de conversion (`float()`, `math.degrees()`, etc.) ■

<sup>a</sup>. Pour information, Paris est à une latitude égale à 48,85341° N, Varsovie 52,22977° N et San Francisco 37,77493° N.

**Exercice 2.11 — Coordonnées polaires.** Écrire un programme qui demande à l'utilisateur les coordonnées cartésiennes d'un point du plan (dans un repère orthonormé) puis qui affiche la conversion en coordonnées polaires de ce même point (voir figure ci-dessous).



Dans un repère  $(O; \vec{i}, \vec{j})$ , on peut repérer un point  $M$  avec ses coordonnées cartésiennes  $M(x; y)$  ou avec des coordonnées polaires  $M(\rho; \theta)$  où  $\rho = OM$  et  $\theta = (\vec{i}, \overrightarrow{OM})$  en radians.

**Indications :** On pourra utiliser une fonction pour calculer la distance entre deux points connaissant leurs coordonnées cartésiennes ainsi que la fonction `math.atan2(y, x)` qui retourne la valeur en radians comprise entre  $-\pi$  et  $\pi$  de l'angle  $(\vec{i}, \overrightarrow{OM})$  où  $M(x; y)$  dans  $(O; \vec{i}, \vec{j})$ . ■

## 3 | Structures conditionnelles

Lors de l'exécution d'un programme, les instructions sont interprétées pas à pas, dans l'ordre où elles ont été écrites : c'est le *flux d'instructions*. Dans ce chapitre, nous allons apprendre à maîtriser ce flux.

### 3.1 Booléens

Il existe un type de variable particulier qui ne peut stocker que deux valeurs possibles. On les appelle des variables *booléennes*<sup>1</sup>. Ces deux valeurs sont représentées par Python par les mots clés `True` ou `False`<sup>2</sup>.

On utilise ces variables pour manipuler le résultats de tests, par exemple des comparaisons :


```
25 >>> 5 > 3
26 True
27 >>> -7 > -3
28 False
```

Les tests disponibles sont :

- plus grand `>`
- plus petit `<`
- plus grand ou égal `>=`
- plus petit ou égal `<=`
- égal `==`
- différent `!=`

En voici la signification et la valeur :

Test	Renvoie <code>True</code> lorsque...
<code>x == y</code>	x est égal à y
<code>x != y</code>	x est différent de y
<code>x &gt; y</code>	x est plus grand que y
<code>x &lt; y</code>	x est plus petit que y
<code>x &gt;= y</code>	x est plus grand ou égal à y
<code>x &lt;= y</code>	x est plus petit ou égal à y
<code>!E</code>	L'expression booléenne E vaut <code>False</code>

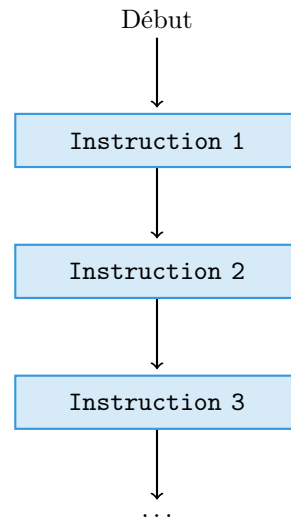
 Ne pas confondre `==` test d'égalité et `=` le signe d'affectation.

1. En mémoire de Georges Boole, mathématicien anglais du XIX<sup>e</sup> qui a imaginé une algèbre qui utilise ce type de variable (voir [https://fr.wikipedia.org/wiki/George\\_Boole](https://fr.wikipedia.org/wiki/George_Boole) pour plus d'informations.)

2. Python interprète aussi 0 comme `False` et 1 comme `True`.



Figure 3.1 – Une séquence de commandes



En Python, c'est le mot clé `if` qui va nous servir à décider si une partie du code va être exécuté ou pas. `if` sera suivi d'une expression booléenne (qu'on mettra entre parenthèses pour plus de lisibilité) et de deux points `:`.

Ensuite, pour faire comprendre à Python quelles tâches doivent être exécutées sous cette condition, on *indente* les lignes de code relatives à ces tâches<sup>3</sup>, c'est-à-dire qu'on insère au début de chaque ligne 4 espaces. On obtient ainsi sous la ligne déclarant la condition, une série de lignes décalées vers la droite : on a défini un *bloc de code* ou *bloc d'instructions*. Toutes les lignes ayant le même retrait font partie du même bloc. Ce bloc de code ne sera lu et exécuté par l'interpréteur que si l'expression booléenne est égale `True`, autrement dit que si la condition est vraie. Dans le cas contraire (l'expression booléenne a pour valeur `False`), ce bloc de code est ignoré par l'interpréteur.

Code	Résultat
<pre> 1 a = 20 2 if (a &gt; 10) : 3     print("a est plus grand que 10") </pre>	a est plus grand que 10

Dans le code ci-dessus, l'exécution de la ligne 2 provoque l'évaluation de l'expression booléenne `a > 10`. Si cette expression est vraie, le bloc qui suit ligne 3 est exécuté. Si elle est fausse, le bloc est ignoré.

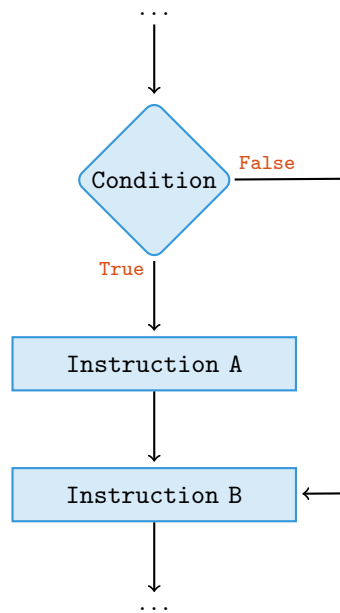
On peut indiquer à Python d'exécuter un autre bloc si le test échoue (voir figure 3.3).

On utilise pour cela l'instruction `else` (suivie de deux points `:`). Le principe est toujours le même, on indente les lignes de codes pour définir le bloc qui devra être exécuté si le test du `if` est faux.

Code	Résultat
<pre> 1 a = 3 2 if (a &gt; 5) : 3     print("a supérieur à 5") 4 else : 5     print("a inférieur ou égal à 5") </pre>	a inférieur ou égal à 5

3. comme pour les fonctions, voir section 1.2 page 7

**Figure 3.2 – Une exécution conditionnelle** : si la condition est vraie, l’instruction A est exécutée, sinon, elle est ignorée.



Dans le code ci-dessus, le test de la ligne 2 échoue : c’est donc le bloc de code sous l’instruction `else` qui est exécuté.

- Exercice 3.2**
1. Écrire une fonction `divisible_par_3` qui détermine si un nombre est divisible par 3. Elle devra retourner `True` si le nombre est divisible par trois et `False` dans le cas contraire.
  2. Écrire un programme qui demande à l’utilisateur un nombre et qui affiche s’il est divisible par trois ou pas. Ce programme devra utiliser la fonction définie à la question précédente.

**Exercice 3.3** Écrire un programme `bizarre.py` qui additionne deux variables `x` et `y` si leur différence est paire et qui les soustrait sinon. Les nombres entiers seront demandés directement à l’utilisateur.

Le résultat sera affiché.

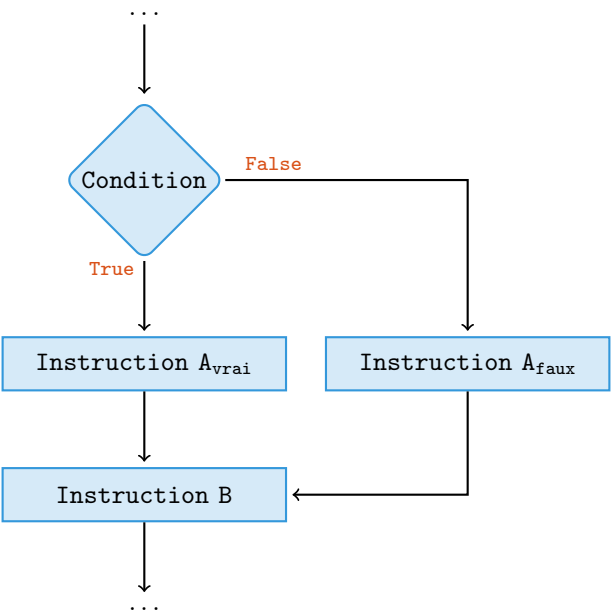
On peut enchaîner les exécutions conditionnelles, c’est-à-dire imbriquer les tests les uns dans les autres (voir figure 3.4).

#### Code

```

1  a = 5
2  if (a > 10) :
3      if (a % 2 == 0) :
4          print "a est un nombre pair plus grand que 10"
5      else :
6          print "a est un nombre impair plus grand que 10"
7  else :
8      if (a % 2 == 0) :
9          print "a est un nombre pair plus petit que 10"
10     else :
11         print "a est un nombre impair plus petit que 10"
  
```

**Figure 3.3 – Une autre exécution conditionnelle** : si la condition est vrai, l'instruction  $A_{\text{vrai}}$  est exécutée, sinon, l'instruction  $A_{\text{faux}}$  est exécutée.



Résultat
a est un nombre impair plus petit que 10

**Exercice 3.4** Une équation du second degré à coefficients réels peut s'écrire sous la forme :

$$ax^2 + bx + c = 0$$

On prouve que cette équation admet soit deux solutions, soit une unique solution, soit aucune solution suivant le signe du *discriminant* de l'équation que l'on calcule avec :

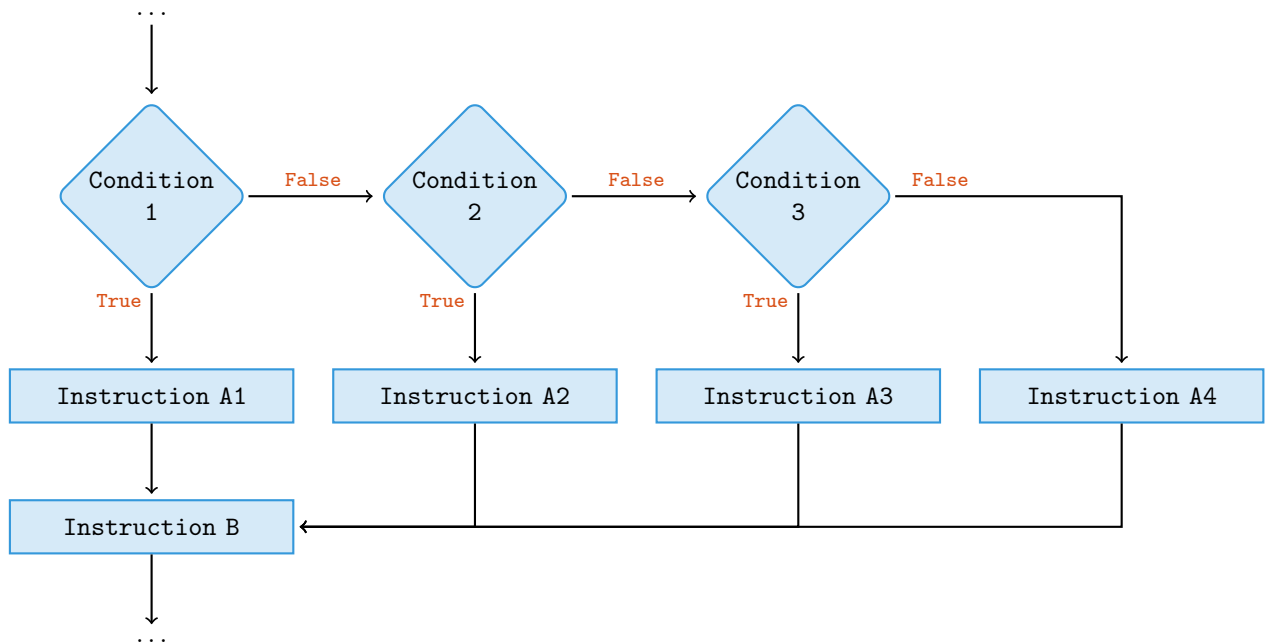
$$\Delta = b^2 - 4ac$$

On peut résumer la situation par le tableau suivant :

Signe de $\Delta$	Solutions
$\Delta > 0$	Deux solutions : $x_1 = \frac{-b + \sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b - \sqrt{\Delta}}{2a}$
$\Delta = 0$	Une seule solution : $x_0 = \frac{-b}{2a}$
$\Delta < 0$	Pas de solution réelle.

Écrire un programme `bynome.py` qui demande à l'utilisateur trois nombres correspondant aux coefficients  $a$ ,  $b$  et  $c$  de l'équation et qui indique les valeurs décimales de la (ou des) solution(s) lorsqu'elles existent. Votre programme devra afficher "Pas de solution réelle" dans le cas où l'équation n'admet pas de solution. ■

**Figure 3.4 – Enchaînement d'exécutions conditionnelles** : si la condition 1 échoue, on teste la condition 2, si elle échoue aussi, on teste la condition 3, etc. (Les instructions  $A_n$  pour  $n > 1$  ne sont pas obligatoires).



**Exercice 3.5** Soit  $f$  une fonction affine par morceaux définie sur  $\mathbb{R}$  par :

$$f(x) = \begin{cases} -2x + 3 & \text{si } x \in ]-\infty; -5] \\ 13 & \text{si } x \in ]-5; 8] \\ x + 5 & \text{si } x \in ]8; +\infty[ \end{cases}$$

Définir une fonction **f** prenant comme paramètre **x** pour la variable  $x$  et qui retourne la valeur de  $f(x)$  suivant celle de  $x$ . ■

### 3.2.3 Répétitions

On peut changer le flux linéaire d'un programme en répétant une instruction ou un bloc d'instructions. On appelle cela une *boucle*. Pour ce faire, on dispose de deux procédés assez différents dans l'approche.

#### Répétition basée sur une condition

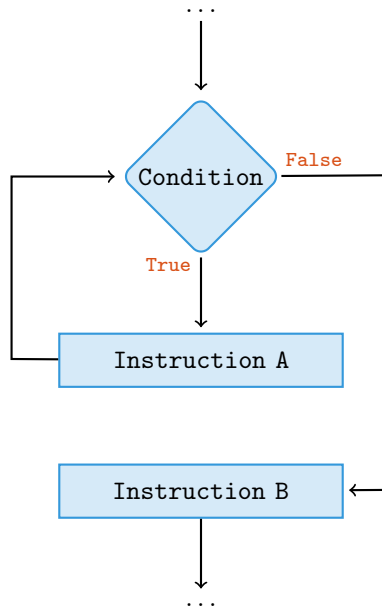
On utilise l'instruction `while`, suivi d'une expression booléenne, suivie de deux points ( `:` ). On désigne comme d'habitude le bloc d'instructions à répéter par des indentations, comme dans le cas des exécutions conditionnelles `if` (voir section 3.2.2 page 20).

Tant que la condition du `while` est vraie (valeur `True`), le bloc est exécuté. Dès que cette même condition devient fausse, le bloc est ignoré et on passe à la suite du programme (voir figure 3.5). Prenons un exemple :

Code	Résultat
<pre> 1 x = 1 2 while (x &lt; 5) : 3     print(x) 4     x = x + 1 </pre>	<pre> 1 2 3 4 </pre>



**Figure 3.5 – Répétition sous condition** : si la condition est vraie, l'instruction A est exécutée et on évalue à nouveau la condition. Lorsque la condition est fausse on continue le programme sans exécuter l'instruction A. *Remarquez que l'instruction A n'est pas chaînée directement avec l'instruction B.*



Examinons le code ci-dessus. À la ligne 1, la variable  $x$  est initialisée. Puis on rencontre l'instruction `while` et Python évalue l'expression booléenne  $x < 5$ . Si cette expression est vraie, Python exécute le bloc d'instruction qui suit : ligne 3, on affiche  $x$  et ligne 4 on ajoute 1 à la variable  $x$  (mise à jour de  $x$ ). Et on revient au test précédent. Tant que l'expression booléenne est vraie, on exécute le bloc d'instructions (lignes 3 et 4).

On parle donc de *boucle* et les lignes 3 et 4 sont appelées *corps de la boucle*. Chaque passage dans la boucle (c'est à dire chaque exécution du corps de la boucle) s'appelle une *itération*.

Il est primordial lorsqu'on demande à Python de faire une boucle de prendre garde à trois choses :

- Initialiser **hors de la boucle** la (ou les) variables du test sinon le test échouera.
- Mettre à jour **dans la boucle** la (ou les) variable(s) du test sinon la variable reste figée et le corps de boucle est exécuté une infinité de fois<sup>4</sup>.
- Bien vérifier que le test peut être évalué comme **False** au cours de l'exécution de la boucle sinon, elle sera répétée une infinité de fois.

**Exercice 3.6** On considère la fonction  $f$  définie sur  $\mathbb{R} \setminus \left\{\frac{1}{2}\right\}$  par  $f : x \mapsto \frac{x+5}{2x-1}$

Écrire un programme Python `tableau.py` qui permettra de remplir ce tableau de valeurs :

$x$	-3,5	-3	-2,5	-2	-1,5	-1	-0,5	0	1	1,5	2	2,5
$f(x)$												

On utilisera une fonction Python nommée `f()` et une boucle dans ce programme.

4. Si vous avez fait une boucle infinie, pas de panique ! Il suffit d'appuyer sur la séquence de touches **Ctrl**+**C** pour arrêter le programme

**Exercice 3.7** Le programme ci-dessous est sensé calculer la somme

$$s = \frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{M}$$

mais il ne fonctionne pas correctement. En réalité, il contient quatre erreurs.

Réécrire le programme afin qu'il affiche le résultat souhaité.

#### Code

```
1 s=0 ; k=1 ; M=100
2 while (k < M)
3     s += 1/k
4 print s
```

**Exercice 3.8** On considère la suite  $(u_n)_{n \in \mathbb{N}}$  définie par

$$\begin{cases} u_0 = 0 \\ u_{n+1} = f(u_n) \text{ pour } n \in \mathbb{N}^* \end{cases}$$

où  $f$  est la fonction définie à l'exercice 3.6.

Écrire un programme `suite.py` qui donne la valeur des premiers termes de la suite pour  $n \geq 10$ .

**Exercice 3.9** On reprend l'exercice 3.2 page 22.

À l'aide d'une boucle, modifier le programme précédent afin qu'une fois affichée la réponse, l'ordinateur demande à l'utilisateur s'il veut recommencer ou pas.

**Exercice 3.10** Pour tout nombre entier  $n$ , on désigne par  $n!$  la *factorielle* du nombre  $n$  qui est définie par :

$$n! = \begin{cases} 1 & \text{pour } n = 0 \\ 1 \times 2 \times \dots \times (n-1) \times n & \text{pour } n > 0 \end{cases}$$

Écrire une programme `factorielle.py` qui demande  $n$  à l'utilisateur et qui affiche ensuite la factorielle du nombre  $n$ .

**Exercice 3.11** Le nombre 1729 est le plus petit nombre qui s'exprime sous la forme d'une somme de deux cubes de deux manières différentes.

Pour vérifier cette assertion, déterminer quatre distincts entiers  $a$ ,  $b$ ,  $c$  et  $d$  tels que

$$a^3 + b^3 = c^3 + d^3$$

On utilisera quatre boucles imbriquées.

**Exercice 3.12 — Premier ou non ?.** Écrire une fonction `isPrime` qui prend un nombre  $N$  en ligne de commande et qui retourne `True` si ce nombre est premier et `False` sinon.

**Indication** – Un nombre premier est un nombre plus grand que 1 qui n'admet pour diviseur que 1 et lui-même. Donc il suffit de prendre les nombres supérieur à 2 et de tester s'il divise  $N$  ou non. On ne peut tester que les nombres inférieur à son carré pour répondre à la question.

**Exercice 3.13 — Décomposition en facteurs premiers.** Écrire un programme qui affiche la décomposition en facteurs premiers d'un nombre entier  $N$  pris en argument dans la ligne de

commandes.

**Indications** – Pour répondre au problème posé, il suffit de commencer par tester la division par 2. Tant que le reste de cette division est nul, on affiche 2 (puisqu'il est un diviseur) et on divise  $N$  par  $i = 2$ . Puis, on ajoute 1 à  $i$  et on recommence, ainsi de suite jusqu'à ce que  $i$  soit inférieur ou égal à  $N/i$ . Si le nombre  $N$  à la sortie de la boucle reste plus grand que 1, cela signifie qu'il est premier et doit être affiché. ■

#### Répétition basée sur le nombre d'itérations

Lorsqu'on connaît à l'avance le nombre d'itérations dans une boucle, on peut utiliser une autre instruction : `for`, assortie de la fonction `range` qui permettra de préciser l'intervalle sur lequel sont basées les itérations. Comme pour l'instruction `while` (et comme toujours), on désigne le bloc d'instructions à répéter par une indentation.

```
5 >>> for x in range(1, 5) :
6 ...     print x
7 ...
8 1
9 2
10 3
11 4
```

On voit que la borne supérieure de l'intervalle est exclue (cela correspond à l'intervalle mathématiques  $[1; 5[$ ).

Si on souhaite commencer à 0, on a pas besoin de préciser la borne inférieure de l'intervalle et pour le coup `range(5)` décide bien de 5 itérations.

```
12 >>> for x in range(5) :
13 ...     print(x)
14 ...
15 0
16 1
17 2
18 3
19 4
```

Un troisième paramètre de la fonction `range()` règle le pas des itérations :

```
20 >>> for x in range(0, 50, 10) :
21 ...     print(x)
22 ...
23 0
24 10
25 20
26 30
27 40
```

**Exercice 3.14** Écrire un programme `moyenne.py` qui demande à l'utilisateur une série de nombre et qui calcule la moyenne de ces nombres. Le programme devra demander en premier lieu à l'utilisateur combien il désire saisir de nombres. La sortie de votre programme pourra ressembler à cela :

## Résultat

```

Combien de nombres souhaitez vous saisir : 0
Je ne peux rien pour vous !
Combien de nombres souhaitez vous saisir : 4
Saisir nombre #1 : 15
Saisir nombre #2 : 9
Saisir nombre #3 : 12
Saisir nombre #4 : 17
La moyenne de ces nombres vaut : 13.25

```

- Exercice 3.15 — Calcul de PGCD.**
1. Écrire une fonction `pgcd_soustractions()` qui prendra deux paramètres `a` et `b` et qui retournera le PGCD de ces deux nombres calculé à l'aide de l'algorithme des soustractions successives.
  2. Écrire une fonction `pgcd_euclide()` qui prendra deux paramètres `a` et `b` et qui retournera le PGCD de ces deux nombres calculé à l'aide de l'algorithme d'Euclide.
  3. Utiliser une des deux fonctions ci-dessus pour écrire une fonction `ppe()` qui prendra deux paramètres `a` et `b` qui retournera `True` si ces deux nombres sont premiers entre eux.

**Exercice 3.16** On rappelle que le *coefficient binomial*  $\binom{n}{k}$  (ou  $C_n^k$ ) donne le nombre de parties de  $k$  éléments dans un ensemble à  $n$  éléments. Sa valeur est donnée par la formule :

$$\binom{n}{k} = C_n^k = \frac{n!}{k!(n-k)!}$$

Écrire une fonction `coef_binomial()` qui prendra deux paramètres `k` et `n` qui retournera la valeur du coefficient binomial correspondant. *On pourra réinvestir le résultat de l'exercice 3.10.*

**Exercice 3.17 — Suite de Fibonacci.** On note  $F_n$  le terme de rang  $n$  de la suite définie par :

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_{n+2} = F_{n+1} + F_n \text{ pour tout } n > 0 \end{cases}$$

Écrire une fonction qui calcule  $F_n$  pour un  $n > 0$  donné.

**Exercice 3.18** À l'aide des boucles, écrire un programme qui affiche les motifs suivants (deux boucles imbriquées pour le motif carré). On écrira un programme par motif.

R sultat

```
*
**
***
****
*****
*****
```

R sultat

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

R sultat

```
*
**
***
****
*****
*****
****
***
**
*
```

R sultat

```

*
**
***
****
*****
****
***
**
*
*
```





## 4 | Listes

### 4.1 Définition

Les *listes* sont des séquences d'objets (c'est-à-dire des nombres, chaînes ou des choses plus complexes définies par l'utilisateur, nous aurons l'occasion d'y revenir). Tous les objets d'une même liste ne doivent pas être forcément du même type<sup>1</sup>.

On déclare une liste en utilisant des crochets (`[ ]`) et en séparant ses éléments par une virgule (`,`).

```
10 mes_entiers = [2, 3, 4, 6, 8, 9]          # liste d'entiers
11 mes_reels = [2.3, 9.7, 4.5, 7.92, 0.002]  # liste de réels
12 mes_chaines = ["blue", "blanc", "rouge"]  # liste de chaînes de caractères
13 pandora_box = [2, "red", 7.89, mes_entiers] # liste d'objets de différents types
```

On a deux manières de créer une liste vide : une paire de crochets vide ou l'instruction `liste`.

```
14 vide1 = list()
15 vide2 = []
```

### 4.2 Utilisation

#### 4.2.1 Accès et affichage

Le mot *séquence* dans la définition du concept de liste implique que les objets ont un certain ordre dans la liste et la place de chaque élément est repérée par un *index*.

**R** Attention : les index commencent à zéro (0) !!!  
Cela signifie que le premier élément de la liste a l'index 0, le deuxième l'index 1 et ainsi de suite.

Pour accéder un élément dans une liste, on se réfère à son index en respectant la syntaxe `liste[index]`.

```
16 >>> liste = ["blue", "blanc", "rouge"]
17 >>> print liste[0]
18 blue
19 >>> print liste[2]
20 rouge
21
```

Attention, si l'index fourni ne correspond pas à un élément de la liste, alors Python renvoie une erreur *list index out of range*.

---

1. Dans d'autres langages, on appellera ce type d'objets des *tableaux* en ce sens qu'on peut accéder à leur contenu par index. Les *listes* seront réservées à des ensembles d'objets en séquences mais sans accès possible par index

```

22 >>> print liste[20]
23 Traceback (most recent call last) :
24   File "<stdin>", line 1, in <module>
25 IndexError : list index out of range
26

```

On peut utiliser des index négatifs pour parcourir une liste en commençant par la fin :

```

27 >>> liste = ["bleu", "blanc", "rouge", "vert", "jaune", "orange"]
28 >>> print liste[-1]
29 orange
30 >>> print liste[-2]
31 jaune
32

```

Pour afficher une liste dans son entier, on utilise la fonction `print` sur la liste elle même. Elle renvoie l'objet liste tel qu'on l'aurait déclaré et ce type d'affichage n'est pas toujours adapté...

```

33 >>> bazard = [2, "red", 5.67, liste]           # une liste hétéroclite !
34 >>> print bazard
35 [2, 'red', 5.67, ['blue', 'blanc', 'rouge']]

```

On peut aussi accéder à une partie de la liste seulement :

```

36 >>> liste = ["bleu", "blanc", "rouge", "vert", "jaune", "orange"]
37 >>> print liste[2:4]           # du 3e (index 2) au 4e (index 4 non compris)
38 ['rouge', 'vert']
39 >>> print liste[3:]           # du 4e a la fin
40 ['vert', 'jaune', 'orange']
41 >>> print liste[:4]           # du début au 4e
42 ['bleu', 'blanc', 'rouge', 'vert']

```

Enfin, on peut accéder par une affectation multiple aux différents éléments d'une liste :

```

43 >>> point = [4, 8]
44 >>> x, y = point
45 >>> print "Abscisse : {}; Ordonnée : {}".format(x, y)
46 Abscisse : 4; Ordonnée : 8

```

On peut connaître la longueur d'une liste en utilisant l'instruction `len` qui renvoie un entier correspondant au nombre d'éléments dans la liste passée en argument.

```

47 >>> liste = [1, 2, 3, 4, 5, 6, 7, 8]
48 >>> print len(liste)
49 8

```

#### 4.2.2 Mutation

En Python, les listes sont des objets *mutables*. Cela signifie que l'on peut changer la valeur des éléments d'une liste sans que cela entraîne la création d'une nouvelle liste.

Pour changer la valeur d'un élément d'une liste, on fait une affectation en bonne et due forme :



```

50 >>> liste=['bleu', 'blanc', 'rouge']
51 >>> print id(liste)
52 4373329824 # identification de la liste
53 >>> liste[0] = 'vert' # on change le premier élément de la liste
54 >>> print liste
55 ['vert', 'blanc', 'rouge']
56 >>> print id(liste)
57 4373329824 # la liste a la même identification

```

En python les listes n'ont pas de longueur fixe. On peut ajouter un élément à une liste ou en insérer.

Pour ajouter un élément à une liste existante, on utilise la fonction `append` qui ajoute un élément à la fin d'une liste.

```

58 >>> print liste
59 ['vert', 'blanc', 'rouge']
60 >>> liste.append('noir')
61 >>> print liste
62 ['vert', 'blanc', 'rouge', 'noir']

```

**Exercice 4.1** Soit une fonction  $f$  définie sur  $\mathbb{R}$  par  $f(x) = 2x^2 - 3x + 1$ . On souhaite dresser un tableau des images par cette fonction  $f$  de toutes les valeurs entières de l'intervalle  $[-3; 6]$ .

1. Écrire la fonction  $f(x)$  qui renvoie les valeurs de  $f(x)$  en fonction de celle de  $x$ .
2. Écrire le code qui permette d'avoir les images souhaitées dans une liste `images`.  
On pourra utiliser une boucle `for` ainsi que la fonction `append()`.

**Exercice 4.2** Reprendre l'exercice 3.13 page 26 mais mettre les facteurs premiers dans une liste `facteurs` et les exposants correspondant dans une autre liste `exposants`.

Pour insérer un objet dans une liste étant donnée son index (et pas sa position !), on utilise la fonction `insert`. Elle prend deux arguments, l'index de l'élément et l'objet à insérer. Par exemple si je souhaite insérer 'rose' en 3<sup>e</sup> position dans la liste, je tape `liste.insert(2, 'rose')` (index 2 = 3<sup>e</sup> position).

```

63 >>> print liste
64 ['vert', 'blanc', 'rouge', 'noir']
65 >>> liste.insert(2, 'rose')
66 >>> print liste
67 ['vert', 'blanc', 'rose', 'rouge', 'noir']

```

Pour supprimer un élément dans une liste j'ai plusieurs solutions. La première consiste à utiliser la commande `del` (ce n'est pas une fonction mais bien une commande). Pour supprimer le deuxième élément de la liste je tape `del liste[1]`.

```

68 >>> print liste
69 ['vert', 'blanc', 'rose', 'rouge', 'noir']
70 >>> del liste[1]
71 >>> print liste
72 ['vert', 'rose', 'rouge', 'noir']

```

On peut aussi utiliser la fonction `remove`. Elle prend en argument un objet à retirer de la liste. Si elle trouve l'objet, elle retire sa première occurrence. Si elle ne le trouve pas, elle renvoie une erreur.

```

73 >>> print liste
74 ['vert', 'rose', 'rouge', 'noir', 'rose']
75 >>> liste.remove('rose')
76 >>> print liste
77 ['vert', 'rouge', 'noir', 'rose']
78 >>> liste.remove('marron')
79 Traceback (most recent call last) :
80   File "<stdin>", line 1, in <module>
81 ValueError : list.remove(x) : x not in list

```

### 4.3 Parcourir une liste

Pour parcourir une liste, j'utilise l'instruction `for` comme indiqué ci-dessous :

```

82 >>> print liste
83 ['vert', 'rouge', 'noir', 'rose']
84 >>> for elt in liste :
85     print elt
86 ...
87 vert
88 rouge
89 noir
90 rose

```

On peut interpréter la ligne 3 comme « Pour chaque élément `elt` de la liste `liste` faire... » et tout ce qui est compris dans le bloc d'instructions après les deux points ( `:` ) est exécuté autant de fois qu'il y a d'éléments dans la liste `liste`.


On peut récupérer l'index de l'élément courant en insérant une deuxième variable (en première position) entre le `for` et le `in` et en utilisant la fonction `enumerate` :

```

91 >>> print liste
92 ['vert', 'rouge', 'noir', 'rose']
93 >>> for index, elt in enumerate(liste) :
94     print "L'élément '{0 :5s}' a pour index {1 :1d}.".format(elt, index)
95 ...
96 L'élément 'vert ' a pour index 0.
97 L'élément 'rouge' a pour index 1.
98 L'élément 'noir ' a pour index 2.
99 L'élément 'rose ' a pour index 3.

```

#### Remarque

 Noter la fonction `range()` (utilisé avec les boucles `for` – voir section 3.2.3 page 27) retourne une liste.

```

100 >>> impairs = range(1, 20, 2)
101 >>> print impairs
102 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

```

### 4.4 Liste en compréhension

Python accepte un mode de construction de listes très facile, très lisible<sup>2</sup> et très efficace (en terme de temps d'exécution) dit *liste en compréhension*. L'idée est de décrire la liste à construire en "filtrant" une autre liste. Pour cela on utilise l'instruction `for`.

2. très *pythonesque*

```

103 >>> puissances_de_2 = [2**k for k in range(10)]
104 >>> print puissances_de_2
105 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

```

On peut éventuellement utiliser des conditions. Voici une première façon de construire une liste des nombres pairs strictement inférieurs à 20.

```

106 >>> pairs = [nb for nb in range(0, 20, 2)]
107 >>> print pairs
108 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

```

En utilisant une condition adjointe dans la définition de la liste, on aurait pu obtenir la liste des nombres entiers pairs inférieur à 20 de cette manière :

```

109 >>> pairs2 = [nb for nb in range(20) if (nb % 2 == 0)]
110 >>> print pairs2
111 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

```

**Exercice 4.3** Reprendre l'exercice 4.1 page 33 en utilisant des listes en compréhension. ■

## 4.5 Exercices

**Exercice 4.4** Soit les listes suivantes :

```

t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Fevrier', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Aout',
      'Septembre', 'Octobre', 'Novembre', 'Decembre']

```

Écrire un programme qui fusionne les listes `t1` et `t2` en une nouvelle liste `t3` en alternant les éléments de 2 et `t1`, soit :

```

['Janvier', 31, 'Fevrier', 28, 'Mars', 31, ...]

```

■

**Exercice 4.5** Toutes les questions qui suivent font référence à la liste ci-dessous :

```
nombres = [32, 5, 12, 8, 3, 75, 2, 15]
```

1. Écrire une fonction `somme` qui prenne une liste de nombres en argument et qui renvoie la somme des éléments de cette liste.  
Tester votre fonction avec la liste `nombres`.
  2. Écrire une fonction `moyenne` qui prenne une liste de nombres en argument et qui renvoie la moyenne de ces nombres.  
Réinvestir la fonction `somme` et tester votre fonction sur la liste `nombres`.
  3. Écrire deux fonctions `maximum` et `minimum` qui prennent une liste de nombres en argument et qui renvoie respectivement le maximum et le minimum de cette liste.  
Tester vos fonctions avec la liste `nombres`.
  4. Écrire un programme qui construit deux listes `pairs` et `impairs` contenant respectivement les nombres pairs et impairs de la liste `nombres`.
- 

**Exercice 4.6 — Fluctuation d'échantillonnage.** Le but de ce programme `stats.py` que vous aller écrire est d'observer le phénomène dit de *fluctuation d'échantillonnage*.

1. À l'aide la construction de listes en compréhension, fabriquer une liste `series` de `N` nombres entiers  $n$  aléatoires<sup>a</sup> tels que

$$1 \leq n \leq 10$$

où `N` sera un entier demandé à l'utilisateur.

2. Construire les listes **effectifs** et **frequences** correspondant respectivement aux effectifs et aux fréquences des nombres de 0 à 10 dans la série **serie**.
3. Présenter vos résultats dans un tableau ressemblant à celui ci-dessous (voir le chapitre 5 pour le formatage d'une chaîne de caractères à l'aide de la fonction **format()** – section 5.3 page 41).

```

112  -----
113  | Nb | Eff | Freq |
114  -----
115  | 1 | 1 | 0.03 |
116  | 2 | 6 | 0.20 |
117  | 3 | 4 | 0.13 |
118  | 4 | 5 | 0.17 |
119  | 5 | 3 | 0.10 |
120  | 6 | 1 | 0.03 |
121  | 7 | 2 | 0.07 |
122  | 8 | 3 | 0.10 |
123  | 9 | 2 | 0.07 |
124  | 10 | 3 | 0.10 |
125  -----

```

4. Faire varier  $N$  et commenter les résultats observés. On pourra adapter le programme pour qu'il prenne en plusieurs arguments en ligne de commande et qu'il affiche un tableau comparatif des résultats pour différentes valeurs de  $N$ .

*a.* On pourra utiliser la fonction **randrange([start,] stop [,step])**. L'argument **stop** est obligatoire et les autres arguments **start** et **step** sont optionnels. Si **stop** est seul renseigné, **randrange** retourne un nombre entier  $x$  tel que  $0 \leq x < \text{stop}$ .

## 5 | Chaînes de caractères

### 5.1 Qu'est-ce qu'une chaîne de caractères ?

#### 5.1.1 Encodage

Avant de présenter les chaînes de caractères, il est important de rappeler que ces chaînes sont représentées par des mots binaires dans la mémoire de l'ordinateur et qu'il existe plusieurs représentations différentes. Python a besoin de savoir quel est l'encodage utilisé dans le script <sup>1</sup>. Pour cela, on introduit en début du document (traditionnellement après le *shebang*, une ligne qui décrit l'encodage utilisé : <sup>2</sup>

```
126 # -*- coding :utf8 -*-
```

Par défaut, Python 2.7 utilise l'encodage *ASCII* et Python 3 l'encodage *UTF-8*. Aujourd'hui, on utilise généralement l'encodage *UTF-8* qui permet de gérer facilement les accents en français et qui sera compris par la plupart des systèmes d'exploitation modernes.

Pour l'anecdote, si vous voulez savoir quels types d'encodage sont reconnus par Python, vous pouvez taper les deux lignes suivantes dans un interpréteur : elle vous donneront la liste de tous ces encodages (et vous pourrez voir qu'il y en a beaucoup !)

```
127 >>> import encodings
128 >>> print(''.join('- ' + e + '\n' for e in
    ↪ sorted(set(encodings.aliases.aliases.values()))))
```

#### 5.1.2 Le type `str`

Sous Python, les chaînes de caractères sont des variables de type `str` (pour *string* en anglais).

```
129 >>> type('abc')
130 <type 'str'>
```

Une chaîne de caractères est représentée au niveau de la mémoire *presque* comme une liste de caractères. On peut accéder à ses éléments un par un, connaître sa longueur à l'aide de la fonction `len()` comme on le ferait d'une "vraie" liste (voir chapitre 4 page 4).

---

1. Si l'encodage utilisé dans le script est différent de l'encodage par défaut utilisé par Python, le script va planter. Donc plutôt que de chercher si ça va marcher ou pas, on intègre **TOUJOURS** la ligne déclarant l'encodage du script

2. Attention à bien respecter la syntaxe, notamment les espaces, sinon l'information ne sera pas prise en compte par Python.

```

131 >>> chaine = "Une petite phrase."
132 >>> print(chaine[0])
133 U
134 >>> print(chaine[-1])
135 .
136 >>> print(chaine[5])
137 e
138 >>> print(len(chaine))
139 18

```

Explication du "presque". Essayons de retrouver les caractères de la chaîne Noël.

```

140 >>> print(chaine)
141 Noël
142 >>> print(len(chaine))
143 5
144 >>> print(chaine[0], chaine[1], chaine[2], chaine[3], chaine[4])
145 N o ? ? l

```

Le mot Noël contient seulement 4 lettres mais pour Python, la longueur de la liste est 5. En réalité, pour coder le caractère ë en Unicode, Python a besoin de deux octets (élément 2 et 3 de la listes). Pour l’affichage, ces octets n’ont de sens que pris en paire mais pas de sens séparément. Donc plutôt qu’une liste de caractères, il vaut mieux concevoir une variable de type **str** comme une liste d’octets encodant des caractères.

On peut préciser à Python en ajoutant une **u** devant la chaîne qu’une chaîne de caractères est une série de code unicode. Chaque chaîne sera alors bien pris comme une séquence de caractères dans leur globalité et non comme une série d’octets. Le type de la chaîne sera alors **unicode** et non **str** comme précédemment.

```

146 >>> chaine = u"Noël"
147 >>> print(len(chaine))
148 4
149 >>> print(chaine[0], chaine[1], chaine[2], chaine[3])
150 N o ë l
151 >>> type(chaine)
152 <type 'unicode'>

```

Des fonctions existent (**encode** et **decode**) pour convertir une chaîne au format **str** en format **unicode** et vice-versa.

Une chaîne de caractères se parcourt malgré tout comme une liste, par exemple à l’aide de la commande **for** :

```

153 >>> chaine = "exemple"
154 >>> for car in chaine :
155     ...     print(car + "*")
156     ...
157 e*
158 x*
159 e*
160 m*
161 p*
162 l*
163 e*

```

**Grosse différence** avec les vraies listes, les chaînes de caractères ne sont pas mutables !

```

164 >>> liste = [1,2,3]
165 >>> print(liste[1])
166 2
167 >>> liste[1] = '*'
168 >>> print(liste[1])
169 *
170 >>> chaine = "man"
171 >>> print(chaine[1])
172 a
173 >>> chaine[1] = 'e'
174 Traceback (most recent call last) :
175   File "<stdin>", line 1, in <module>
176   TypeError : 'str' object does not support item assignment

```

## 5.2 Manipulations

### 5.2.1 Sous-chaînes

Python possède une technique unique (appelée *slicing*) qui permet de récupérer un morceau de chaîne (sous-chaîne). La syntaxe est la suivante : supposons que ma chaîne s'appelle `ch`, je désigne une sous-chaîne de `ch` par `ch[n :m]` où `n` est l'index du premier caractère pris dans la sous-chaîne et `m` l'index de la lettre *suivant* celle où on s'arrête.

```

177 >>> chaine = "Prolifique"
178 >>> print(chaine[0 :3])
179 Pro
180 >>> print(chaine[ :3])
181 Pro
182 >>> print(chaine[3 :])
183 lifique

```

Si rien n'est inscrit à la place de `n` dans `chaine[n :m]`, alors Python considère qu'il doit commencer au premier caractère. Si rien n'est inscrit à la place de `m` dans `chaine[n :m]`, alors Python considère qu'il doit terminer la sous-chaîne avec le dernier caractère.

### 5.2.2 Concaténation

La *concaténation* de deux chaînes de caractères est la mise bout à bout de ces deux chaînes. Sous Python, on utilise le symbole d'addition `+` pour cette opération.

```

184 >>> a = "Hello,"
185 >>> b = " world !"
186 >>> print(a + b)
187 Hello, world !

```

### 5.2.3 Répétition

Si l'on veut répéter plusieurs fois la même chaîne de caractères, on utilise l'opérateur `*`.

```

188 >>> print('-', * 20)
189 -----
190 >>> c = "Pipo"
191 >>> print(c * 12)
192 PipoPipoPipoPipoPipoPipoPipoPipoPipoPipoPipo

```

### 5.2.4 Comparaison

Les chaînes de caractères sont comparables au même titre que des nombres.

```
193 >>> print("arbre" < "arbuste")
194 True
195 >>> print("arbre" < "arbousier")
196 False
```

La comparaison on le voit ne se fait pas sur les longueurs de chaînes. L'ordre qui prévaut dans ces comparaisons de chaînes est l'ordre de déclaration des caractères dans le code ASCII qui est l'ordre alphabétique. Ainsi la chaîne "arbre" est inférieure à la chaîne "arbuste" car le caractère 'r' est avant le caractère 'u' dans l'ordre alphabétique mais cette même chaîne est supérieure à la chaîne "arbousier" car le caractère 'r' est après le caractère 'o'. On peut donc se servir de cette comparaison de chaînes de caractères pour classer des mots dans l'ordre alphabétique.

**R Attention !** Les minuscules venant après les majuscules dans la table des caractères ASCII, cela peut conduire à des résultats bizarres :

```
197 >>> print('arbre' < 'Type')
198 False
199 >>> print('arbre' < 'type')
200 True
```

### 5.2.5 Chaînes en tant qu'objets

Les chaînes de caractères sont aussi considérées comme des *objets*<sup>3</sup> par Python et à ce titre, Python propose un grand nombre de fonctions opérant sur des chaînes de caractères.

En voici quelques exemples :

Nom	Fonction
<code>lower</code>	convertit une chaîne en minuscule
<code>upper</code>	convertit une chaîne en majuscule
<code>title</code>	convertit l'initiale de chaque mot en majuscule
<code>capitalize</code>	convertit la première lettre de chaque ligne en majuscule
<code>swapcase</code>	change les majuscules en minuscules et vice-versa
<code>replace(c1,c2)</code>	remplace tous les caractères c1 de la chaîne par le caractère c2
<code>count(s)</code>	compte le nombre d'apparition d'une sous-chaîne s dans la chaîne
<code>find(s)</code>	cherche la position de la sous-chaîne s dans la chaîne
<code>split</code>	convertit la chaîne en une liste de sous-chaînes en utilisant le caractère espace pour caractère de séparation (ou tout autre caractère passé en argument)
<code>join</code>	prend une liste de chaînes en argument et renvoie une autre chaîne en intercalant la chaîne appelante

Quand une fonction intervient sur un objet, on la place *après* l'objet précédée par un point : `objet.fonction(arg1, arg2, ...)`.

3. Nous reviendrons plus en détails plus tard sur la notion d'objet...



```

201 >>> chaine = "jérémy"
202 >>> print(chaine.upper())
203 JÉRÉMY
204 >>> print(chaine.capitalize())
205 Jérémy
206 >>> print(chaine.count(u'é'))
207 2
208 >>> print(chaine.replace(u'é', '$'))
209 j$ r$ my

```

### 5.3 Formater une chaîne

L'instruction `format` permet de formater une chaîne de caractères. Prenons un exemple :

```

210 >>> x = 2
211 >>> result = "La racine carrée de {0} est {1}".format(x, math.sqrt(x))
212 >>> print(result)
213 La racine carrée de 2 est 1.41421356237

```

À l'intérieur de la chaîne `result` il y a deux *champs*. Cette fonction prend autant d'arguments que la chaîne contient de champs, dans l'ordre où ils sont énoncés. Ils peuvent être nommés pour plus de lisibilité : dans ce cas, on utilise ces noms pour passer les valeurs aux arguments de la fonction.

```

214 >>> x = 2
215 >>> result = "La racine carrée de {nombre} est {racine}".format(nombre = x,
216 ↪ racine = math.sqrt(x))
217 >>> print(result)
218 La racine carrée de 2 est 1.41421356237

```

On peut préciser après le champ le type attendu :

```

218 >>> x = 2
219 >>> result = "La racine carrée de {nombre :s} est {racine}".format(nombre = "x",
220 ↪ racine = math.sqrt(x))
221 >>> print(result)
222 La racine carrée de x est 1.41421356237

```

Dans l'exemple ci-dessus, j'ai précisé que le type du champ `nombre` devait être une chaîne de caractère (`{nombre :s}`). La syntaxe du type de format est la suivante :

[[*car*]*align*][*largeur*][*précision*]*type*

**align** Détaille l'affichage du champ.

+	Affiche le signe + avant les nombres positifs (omis par défaut)
<	Aligne le champ sur la gauche
>	Aligne sur la droite
^	Centré dans le champ
car	Utilise le caractère <car> à la place des espaces pour couvrir la largeur du champ si elle est spécifiée (ceci est optionnel, espace par défaut).

**largeur** Minimum de caractères à afficher dans le champ. Si le nombre de caractères du paramètre est supérieur alors ce nombre n'est pas respecté.

**precision** Maximum de chiffres après la virgule pour un décimal. Maximum de caractères à afficher dans le champ pour une chaîne. *Noter le point avant cette valeur.*

**type** type du champ

<b>d</b>	Nombre entier
<b>f</b>	Nombre décimal
<b>%</b>	Pourcentage
<b>,</b>	Affiche les nombre à l'américaine (avec des virgule pour séparer les classes de mille)
<b>s</b>	Chaîne de caractères

Quelques exemples :

**Exemple 5.1** Un premier exemple où on dresse une liste de nombres et de leurs racines carrées arrondies au millième près.

#### Code

```
1 >>> i = 5
2 >>> while (i < 12) :
3 ...     print("Nombre : {nb :2d}, racine carrée : {racine :2.3f}".format(nb =
  ↳ i, racine = math.sqrt(i)))
4 ...     i += 1
```

#### Résultat

```
Nombre : 5, racine carrée : 2.236
Nombre : 6, racine carrée : 2.449
Nombre : 7, racine carrée : 2.646
Nombre : 8, racine carrée : 2.828
Nombre : 9, racine carrée : 3.000
Nombre : 10, racine carrée : 3.162
Nombre : 11, racine carrée : 3.317
```

**Exemple 5.2** Dans cet autre exemple, on dresse un tableau des carrés, des cubes et des puissances quatrièmes des 10 premiers entiers.

#### Code

```
1 print("-----")
2 print("| x | x^2 | x^3 | x^4 |")
3 print("-----")
4
5 i = 1
6 result = "| {x :<2d} | {carre :<3d} | {cube :<4d} | {quatre :<5d} |"
7 while (i <= 10) :
8     print(result.format(x=i, carre=i*i, cube=i*i*i, quatre=i*i*i*i))
9     i += 1
10 print("-----")
```

## Résultat

x	x <sup>2</sup>	x <sup>3</sup>	x <sup>4</sup>
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000

**Exemple 5.3** Dans ce dernier exemple, on écrit une résultat sous la forme d'un pourcentage arrondi au centième.

## Code

```
1 >>> x = 8769.23
2 >>> y = 78817.98
3 >>> print("Un pourcentage : {0 :.2%}".format(x/y))
```

## Résultat

Un pourcentage : 11.13%

## 5.4 Exercices

Les exercices suivants pourront être résolus *d'abord* sans utiliser les fonctions intégrées.

**Exercice 5.1** Écrire une fonction `contientE` qui prend une chaîne de caractères en argument et qui renvoie `True` si cette chaîne de caractères contient la lettre `e` et `False` sinon. ■

**Exercice 5.2** Écrire une fonction `compteA` une fonction qui compte le nombre de A (majuscule ou minuscule) dans une chaîne de caractères passée en argument. ■

**Exercice 5.3** Écrire une fonction qui prend une chaîne de caractères quelconque et qui intercale le caractère `*` entre toutes les lettres et renvoie cette chaîne transformée. Par exemple `ordinateur` deviendra `o*r*d*i*n*a*t*e*u*r`. ■

**Exercice 5.4** Écrire une fonction `trouvePosition` qui prend deux arguments : une chaîne de caractère et une caractère seul (chaîne de longueur 1). Elle retournera un entier égal à l'index de la première occurrence du caractère dans la chaîne ou `-1` si le caractère n'existe pas dans la chaîne. ■

**Exercice 5.5** Écrire une fonction `inverseChaine` qui prend une chaîne de caractère en argument et qui retourne une chaîne égale aux caractères de l'argument dans l'ordre inverse. Par exemple, pour `normal`, la fonction retournera `lamron`. ■

**Exercice 5.6 — Palindromes.** Écrire une fonction `isPalindrome` qui détermine si une chaîne de caractères est un palindrome ou non.

Un palindrome est une chaîne qui peut se lire dans les deux sens. Par exemple `Hannah` ou `Engage le jeu que je le gagne` sont des palindromes (remarquez que l'on ne tient pas compte du fait qu'une lettre est une majuscule ou pas).

**Exercice 5.7** Écrire un programme qui prenne un entier en ligne de commandes et qui affiche les les puissances de 2 entières qui sont plus petites que ce nombre.

**Exercice 5.8** Écrire un programme qui affiche une table formatée des nombres suivants :  $n$ ,  $n^2$ ,  $n \ln n$ ,  $2^n$  et  $e^n$ . Prendre deux chiffres après la virgule pour les valeurs décimales et aligner les nombres à droite.

**Exercice 5.9** Écrire un programme qui affiche dans une table formatée, les valeurs des cosinus, sinus et tangente des angles suivants :  $0$ ,  $\pi$ ,  $\frac{\pi}{2}$ ,  $\frac{\pi}{3}$ ,  $\frac{\pi}{4}$ ,  $\frac{\pi}{5}$ ,  $\frac{\pi}{6}$ . On donnera des valeurs approchées à  $10^{-3}$  près.

**Exercice 5.10** Écrire une fonction `table` qui admet un paramètre `nombre` et qui affiche de manière formatée la table de multiplication du nombre passé en argument.

```

2 0 x 7 = 0
3 1 x 7 = 7
4 2 x 7 = 14
5 3 x 7 = 21
6 4 x 7 = 28
7 5 x 7 = 35
8 6 x 7 = 42
9 7 x 7 = 49
10 8 x 7 = 56
11 9 x 7 = 63
12 10 x 7 = 70

```

# A | Nombres complexes

Python sait calculer avec des nombres complexes. Nous allons voir dans cette partie comment.

## A.1 Implémentation native

Sans librairie particulière, Python connaît les nombres complexes et sait calculer avec.

### A.1.1 Définition

On utilise la lettre `j` dans la définition du nombre à la place du  $i$  utilisé en mathématiques <sup>1</sup>. Par exemple, le nombre  $3i$  s'écrira `3j`.

```
13 >>> a = 3j
14 >>> type(a)
15 <class 'complex'>
```

**R** Attention ! `j` ne s'utilise pas tout seul. En effet, il pourrait être confondu avec la variable du même nom.

```
16 >>> b = 3 + j
17 Traceback (most recent call last) :
18   File "<stdin>", line 1, in <module>
19   NameError : name 'j' is not defined
20 >>> b = 3 + 1j
```

### A.1.2 Parties réelles et imaginaires

On peut retrouver les parties réelles et imaginaires d'un nombre complexe : Python les conçoit comme des *attributs* du nombre complexe créé et les nomme `real` et `imag` :

```
21 >>> a = -2 + 5j
22 >>> a.real
23 -2.0
24 >>> a.imag
25 5.0
```

On peut aussi déterminer le *conjugué* d'un nombre complexe à l'aide de la fonction `conjugate()` :

```
26 >>> a = -2 + 5j
27 >>> a.conjugate()
28 (-2-5j)
```

Pour déterminer le *module* d'un nombre complexe on utilise la fonction `abs()` :

---

1. Les physiciens font de même.

```

29 >>> a = 1 + 1j
30 >>> abs(a)
31 1.4142135623730951

```

**Exercice A.1** Vérifier sur des exemples que si  $u = a + ib$  alors  $|u| = \sqrt{a^2 + b^2}$ . ■

### A.1.3 Opérations

On peut effectuer les quatre opérations courantes :

```

32 >>> a = 3 - 2j
33 >>> b = 3 + 1j
34 >>> a + b
35 (6-1j)
36 >>> a - b
37 -3j
38 >>> a * b
39 (11-3j)
40 >>> a / b
41 (0.7000000000000001-0.8999999999999999j)

```

**R** Python effectue les conversions si besoin est :

```

42 >>> a = 3 - 2j
43 >>> c = 4
44 >>> type(a) ; type(c)
45 <class 'complex'>
46 <class 'int'>
47 >>> a + c # Pas d'erreur !
48 (7-2j)

```

**Exercice A.2** Vérifier sur des exemples que si  $u = a + ib$  et  $v = c + id$  alors on a les formules suivantes :

$$uv = (ac - bd) + i(bc + ad)$$

$$\frac{u}{v} = \frac{ac + bd}{c^2 + d^2} + i \frac{bc - ad}{c^2 + d^2}$$

On peut aussi calculer des puissances avec ces nombres complexes :

```

49 >>> a = 3 - 2j
50 >>> a ** 2
51 (5-12j)

```

On peut tenter de trouver la racine carrée d'une puissance utilisant l'opérateur `**` mais on n'obtient qu'une des deux racines carrées :

```

52 >>> a ** 0.5
53 (1.8173540210239707-0.5502505227003375j)

```

## A.2 Librairie cmath

L'utilisation de la librairie `cmath` est moins intuitive mais permet des calculs plus... complexes ! On va surtout pouvoir accéder à la forme trigonométrique des nombres complexes.

On charge la librairie `cmath` comme une librairie usuelle<sup>2</sup> :

```
54 >>> import cmath
```

### A.2.1 Forme trigonométrique

Lorsqu'on a saisi un nombre complexe à l'aide de ce que nous avons vu section A.1 comme par exemple `a = 1 + 1j`, alors on va pouvoir grâce à la fonction `polar()` accéder à sa forme trigonométrique : cette fonction renvoie le *module* et l'*argument* (en radian) correspondant.

```
55 >>> a = 1 + 1j
56 >>> cmath.polar(a)
57 (1.4142135623730951, 0.7853981633974483)
```

On peut avoir accès directement au module et à l'argument à l'aide des fonctions `abs()` (déjà vu ci-dessus) et `phase()` :

```
58 >>> abs(a)
59 1.4142135623730951 # module
60 >>> cmath.phase(a)
61 0.7853981633974483 # argument (en radians)
62 >>> import math
63 >>> math.degrees(cmath.phase(a)) # pour avoir l'argument en degrés
64 45.0
```

On peut définir un nombre complexe à partir de son argument et de son module à l'aide de la fonction `rect()`.

```
65 >>> cmath.rect(4, math.pi/3)
66 (2.0000000000000004+3.4641016151377544j)
```

### A.2.2 Fonctions spécifiques

Le module `cmath` dispose d'une fonction racine carrée `sqrt()` plus évoluée que le module `math`.

```
67 >>> math.sqrt(-3)
68 Traceback (most recent call last) :
69   File "<stdin>", line 1, in <module>
70   ValueError: math domain error # calcul impossible avec 'math'
71 >>> cmath.sqrt(-3)
72 1.7320508075688772j # pas de problème avec 'cmath'
```

Le module `cmath` dispose d'une exponentielle complexe. Un exemple avec  $e^{\pi i} = -1$  :

```
73 >>> cmath.exp(cmath.pi*1j)
74 (-1+1.2246467991473532e-16j) # correct... à une approximation près !
```

On voit qu'on a un problème d'approximation dans les calculs. Le module `cmath` possède une fonction `isclose()` qui renvoie `True` si deux nombres sont proches l'un de l'autre. Cela est bien utilisé pour les tests. On peut régler la tolérance à l'aide d'un argument supplémentaire (voir la documentation<sup>3</sup> pour plus d'informations).

2. elle est présente par défaut, par besoin d'installation supplémentaire

3. <https://docs.python.org/3.5/library/cmath.html#classification-functions>

```
75 >>> a = cmath.exp(cmath.pi*1j)
76 >>> cmath.isclose(a, -1)
77 True
78 >>> a == -1
79 False
```

**Exercice A.3** Vérifier sur des exemples que  $e^{ix} = \cos x + i \sin x$ . ■



# Index Général

## A

Adresse mémoire ..... 11

## B

Bloc ..... 21

Boole, George ..... 19

Boucle ..... 24

## C

Complexes

    Argument ..... 47

    Conjugué ..... 45

    Définition ..... 45

    Imaginaire, Partie ..... 45

    Module ..... 45

    Réelle, Partie ..... 45

Concaténation ..... 39

## D

Divisions (2 types) ..... 5

## F

Fonctions

    Definitions ..... 7

    Mathématiques ..... 6

    Paramètres ..... 8

    Signature ..... 7

## I

Indentation ..... 21

Itération ..... 25

## O

Opérateurs booléens ..... 20

## S

Shebang ..... 37

Slicing ..... 39

Sous-chaîne ..... 39

## T

Tables de vérité ..... 20

Type (variable) ..... 11

## V

Variable ..... 11



# Index des Commandes

On liste ici toutes les commandes, mots clés, fonctions, modules...

## Symbols

*	7, 39	if	21
,	9	imag	45
:	7	import	6
<	19	input()	16
<=	19	insert	33
=	11, 13	int()	15
==	19	isclose()	47
>	19		
>=	19	<b>J</b>	
#	16	j	45
		join	40
<b>A</b>		<b>L</b>	
abs()	45	len	32
and	20	liste	31
append	33	lower	40
<b>C</b>		<b>M</b>	
capitalize	40	math	6
cmath	46	math.cos()	6
conjugate()	45	math.pi	6, 8
count	40	math.sqrt()	6, 8
<b>D</b>		<b>N</b>	
def	7	not	20
del	33	<b>O</b>	
<b>E</b>		oct()	15
else	21	or	20
enumerate	34	<b>P</b>	
<b>F</b>		phase()	47
False	19	polar()	47
find	40	print()	12
float()	15	<b>R</b>	
for	27	range	27
from	6	real	45
<b>H</b>		rect()	47
hex()	15	remove	33
<b>I</b>		replace	40
id()	11	resultat()	8
		return	7

**S**

split ..... 40  
sqrt() ..... 6, 47  
str() ..... 15  
swapcase ..... 40

**T**

title ..... 40

True ..... 19  
type() ..... 11

**U**

upper ..... 40

**W**

while ..... 24, 25