

## Plan du chapitre

<b>I Les types simples</b>	<b>1</b>
I.1 Opérations sur les entiers et sur les flottants . . . . .	1
I.2 Les booléens . . . . .	2
<b>II Les variables</b>	<b>5</b>
II.1 La notion de variable . . . . .	5
II.2 Etat et valeur d'une expression . . . . .	6
II.3 Déclaration et initialisation . . . . .	6
II.4 Affectation . . . . .	7
II.5 Echanger le contenu de deux variables . . . . .	8
<b>III Les types composés</b>	<b>10</b>
III.1 Les $n$ -uplets . . . . .	10
III.2 Chaînes de caractères . . . . .	12
III.3 Listes . . . . .	16
III.4 Conversions . . . . .	16
<b>IV Trois exercices corrigés</b>	<b>17</b>

---



Le but de ce cours est d'apprendre à manipuler des valeurs avec Python, c'est-à-dire :

- d'une part comment mener des calculs sur des valeurs (nombres, booléens, chaînes de caractères, listes...)
- et d'autre part, comment les stocker dans des variables.

Nous utilisons l'interpréteur interactif Python pour illustrer ces notions.

## I Les types simples

Le type d'une valeur correspond à la nature de celle-ci. On peut l'obtenir grâce à **type**.

Les trois types que nous étudions dans ce paragraphe sont les entiers, les flottants et les booléens.

```
1 >>> type(45)
2 <class 'int'>
3 >>> type(45.1)
4 <class 'float'>
5 >>> type(4==8)
6 <class 'bool'>
```

### I.1 Opérations sur les entiers et sur les flottants

Les quatre opérations de base se font de manière simple sur les entiers et les flottants :

```
1 >>> 2-(4-3)
2 1
3 >>> 1.2*2
4 2.4
5 >>> 1.0/3.0
6 0.3333333333333333
```

Lors d'opérations entre entiers et flottants, les entiers sont convertis à la volée en flottants :

```
1 >>> 1.2+2
2 3.2
```

L'opérateur puissance utilise le symbole **\*\***; pour obtenir le reste d'une division euclidienne, on utilise le symbole modulo **%** :

```
1 >>> 2**3
2 8
3 >>> 2**100
4 1267650600228229401496703205376
5 >>> 2.0**100.0
```

```
6 1.2676506002282294e+30
7 >>> 12%5
8 2
9 >>> -12%5
10 3
```



### Savoir-faire : convertir une expression numérique d'un type à un autre

Pour utiliser le nombre 12, on écrira **12** si l'on veut utiliser un entier, et **12.0** ou simplement **12.** si l'on veut un flottant.

On peut si besoin convertir un entier en flottant en lui appliquant la fonction **float**, et un flottant en un entier par la fonction **int**. Attention : **int(x)** ne calcule pas la partie entière de **x**, mais le tronque, c'est-à-dire calcule la partie entière de  $|x|$ , puis affecte le résultat du signe de **x**.

```
1 >>> float(4)
2 4.0
3 >>> int(9.1)
4 9
5 >>> int(-8.4)
6 -8
```

## I.2 Les booléens

### a Les deux valeurs **True** et **False**

Les booléens constituent un type spécial dont les valeurs sont particulièrement simples : il n'y en a que deux, **True** et **False**. Ils servent à représenter le résultat de l'évaluation d'expressions logiques qui peuvent prendre soit la valeur vraie, représentée par **True**, soit la valeur fausse, représentée par **False**.

### b Opérateurs sur les booléens

Les opérateurs sur les booléens correspondent aux connecteurs logiques que l'on manipule en mathématiques ou en sciences de l'ingénieur :

- la négation, dont le symbole Python est **not** : l'expression **not b** a la valeur **True** si **b** s'évalue à **False**, et la valeur **False** si **b** s'évalue à **True**.
- La conjonction, i.e. le *et logique*, dont le symbole Python est **and** : l'expression **b1 and b2** a la valeur **True** si **b1** et **b2** s'évaluent à **True**. Si l'une au moins des deux expressions **b1** et **b2** s'évalue à **False**, alors **b1 and b2** a la valeur **False**.

- La disjonction, i.e. le *ou logique*, dont le symbole Python est **or** : l'expression **b1 or b2** a la valeur **False** si les deux expressions **b1** et **b2** ont la valeur **False**. Si l'une au moins des deux expressions **b1** et **b2** s'évalue à **True**, alors **b1 or b2** a la valeur **True**.

Voici quelques exemples :

```
1 >>> True and False
2 False
3 >>> True and True
4 True
5 >>> not True
6 False
7 >>> not (not True)
8 True
```

Les opérateurs **and** et **or** sont dits paresseux : ils ne calculent que ce qui est nécessaire pour évaluer une expression. Par exemple :

```
1 >>> 0!=0 and 1/0==2
2 False
```

Le booléen situé à gauche de **and** vaut **False**, celui de droite n'est pas évalué. Ecrite dans l'ordre inverse, l'expression produit une erreur de division par 0 :

```
1 >>> 1/0==2 and 0!=0
2 ZeroDivisionError: division by zero
```

De même, l'opérateur **or** n'évalue pas son membre droit si son membre de gauche vaut **True**.

### c Opérateurs de comparaison

L'apparition la plus fréquente de booléens se fait lors de comparaisons d'expressions d'autres types.

La comparaison la plus élémentaire est le test d'égalité. L'expression **e1==e2** s'évalue au booléen **True** si **e1** et **e2** s'évaluent à des valeurs égales, sinon elle s'évalue à **False**.

```
1 >>> 1==3-2
2 True
3 >>> 1==0
4 False
```

Python dispose d'un raccourci pour l'expression **not (e1==e2)**, à savoir **e1 !=e2**.

**⚠ Attention danger ! L'égalité et les flottants**

~~~~~ C'est l'occasion de constater que, conformément à ce qui a été vu lors des cours précédents, les flottants ne sont pas des représentations exactes des nombres décimaux.

```
1 >>> 0.1+0.1+0.1==0.3
2 False
```

**⚠ Attention danger !**

~~~~~ Tester l'égalité de deux flottants est presque toujours une erreur. Il vaut mieux tester si la valeur absolue de la différence de ces deux flottants est ou non significative, c'est-à-dire est plus petite qu'une précision donnée.

Les types que nous allons rencontrer sont pour la plupart également comparables pour une relation d'ordre fixée. On peut alors utiliser cet ordre pour comparer deux expressions :

```
1 >>> 1<7
2 True
3 >>> 1<=1
4 True
5 >>> 1<2<3
6 True
```

**Entrainement 1**

Donner la valeurs des booléens suivants :

1.  $3*3.5>10$
2.  $3.*7==21$
3.  $3-1>=1$  or  $8==7$
4.  $3-1=>1$
5.  $\text{not}(2-1==1==4+3)$
6.  $(5*2==10$  or  $7-8!=1)$  and  $(3\%2==1)$

Réponses :

```
1 >>> 3*3.5>10
2 True
3 >>> 3.*7==21
4 True
```

```
5 >>> 3-1>=1 or 8==7
6 True
7 >>> 3-1=>1
8 SyntaxError: invalid syntax
9 >>> not(2-1==1==4+3)
10 True
11 >>> (5*2==10 or 7-8!=1) and (3%2==1)
12 True
```

### Entrainement 2

Ecrire des expressions booléennes traduisant les conditions suivantes. Les nombres mentionnés sont tous des entiers.

1. L'entier **n** est divisible par 5.
2. Les entiers **m** et **n** sont de même signe.
3. Les trois entiers **m,n** et **p** sont de même signe.
4. **n** est le plus petit multiple de 7 strictement supérieur à  $10^{24}$ .

Réponses :

1.  $n\%5==0$
2.  $n*m>=0$
3.  $(m*n>=0) \text{ and } (m*p>=0) \text{ and } (n*p>=0)$
4.  $(n\%7==0) \text{ and } (n>10^{24}) \text{ and } (n-7<=10^{24})$

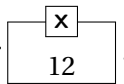
## II Les variables

### II.1 La notion de variable

#### Définition

Une *variable informatique* est une partie de la mémoire de l'ordinateur, désignée par un nom, et qui peut stocker une valeur (entier, flottant, booléen, chaîne de caractères, *n*-uplet, liste...).

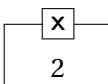
Pour faire référence à une zone mémoire, on utilise un nom de variable. On représentera dans ce cours une variable par un diagramme en forme de rectangle dont le contenu est la valeur de la variable, surmonté d'une étiquette figurant le nom de la variable.

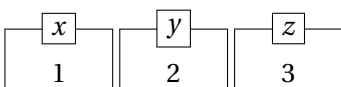
Ainsi, la variable nommée  $x$  de contenu 12 sera représentée par .

## II.2 Etat et valeur d'une expression

L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé l'*état*.

De l'état courant dépend donc la valeur de toute expression contenant au moins un nom de variable. Lors de l'évaluation de l'expression, on remplace les noms de variables par leurs valeurs; on dit qu'on a substitué à la variable sa valeur dans l'expression.

L'expression  $x+3$  prend ainsi la valeur 5 dans l'état , car le nom de la variable  $x$  a été remplacé par son contenu 2.

Dans l'état  qui comporte plusieurs variables, l'expression  $x+y*z+z*z$  s'évalue en  $1+2*3+3*3$  et donc a la valeur 16.

Au cours de l'évaluation d'une expression, l'état ne change pas. Ainsi, dans l'évaluation précédente, la même valeur sera substituée aux trois occurrences du nom  $z$ .

Si, lors de l'évaluation d'une expression, un nom de variable est utilisé alors qu'il n'apparaît pas dans l'état courant, il devient impossible d'attribuer une valeur à cette expression. Python renvoie alors le message suivant :

```
1 >>> t+5
2 NameError: name 't' is not defined
```

## II.3 Déclaration et initialisation

### Définition

*Déclarer* une variable consiste à l'ajouter à l'état. En Python, cette déclaration est toujours accompagnée d'une initialisation.

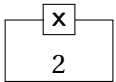
Le nom d'une variable peut être constitué uniquement de lettres, de chiffres, et du caractère «`_`» (underscore), et doit nécessairement commencer par une lettre; les lettres majuscules et minuscules sont distinguées. Le nom d'une variable doit être évidemment différent des mots appartenant à la syntaxe de Python (**print**, **def**, **if**, etc.)

Une fois le nom choisi, la variable est déclarée en évaluant une *instruction* de la forme :

```
1 nom_de_la_variable=expression
```

Une instruction est une forme d'interaction qui demande d'effectuer une modification de l'état. En général, une instruction n'a pas de valeur; après l'avoir exécutée, Python n'affiche rien. Par exemple :

```
1 x=2
```

redonne directement l'invite de l'interpréteur, non sans avoir modifié l'état qui contient désormais .

Il suffit d'évaluer une expression à la suite pour le constater :

```
1 >>> x+3
2 5
```

L'instruction :

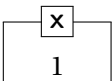
```
1 >>> y=x
```

fait passer dans l'état . En effet, l'expression  $x$  a été évaluée à 2, et c'est le résultat de cette évaluation qui est placé dans la variable de nom  $y$ .

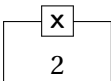
On remarque qu'il n'est pas possible de supprimer une déclaration de l'état. Pour que  $x$  ne soit plus défini, il n'y a pas d'autre solution que de relancer Python.

## II.4 Affectation

Pour changer la valeur d'une variable, on utilise la même instruction que pour la déclaration.

Par exemple, si on est dans l'état , l'instruction :

```
1 >>> x=2
```

fait passer dans l'état .

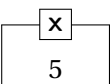
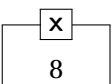
### Attention danger !

En Python, le symbole « = » n'a pas la même signification qu'en mathématiques. Alors qu'en mathématiques, il signifie « est égal à », en Python il signifie « est affecté à la valeur ».

On peut alors imaginer des expressions qui seraient absurdes en mathématiques, mais qui ne posent aucun problème en Python, comme par exemple :

```
1 >>> x=x+3
```

Dans cette instruction,  $x$  joue deux rôles différents : à gauche, il s'agit du nom de la variable sur laquelle s'effectue l'affectation. A droite,  $x$  est un élément de l'expression qui va être évaluée.

Donc, si on est dans l'état  avant cette affectation, cette instruction nous fera passer dans l'état .

Ce genre d'opération est très souvent utilisé en programmation, par exemple lorsque l'on souhaite incrémenter un compteur dans un algorithme.

Vérifions à l'aide de la console :



```

1 >>> x=5
2 >>> x=x+3
3 >>> x
4 8

```

### Savoir-faire

L'instruction  $x=x+1$  est équivalente à l'instruction  $x+=1$ , qu'on peut lire « ajouter 1 à  $x$ , sous entendu dans l'état courant. Des opérations-affectations similaires existent pour la plupart des opérateurs courants : par exemple,  $y*=e$  multiplie la variable  $y$  par l'expression  $e$ , et  $z-=e$  retranche l'expression  $e$  à la variable  $z$ .

Il existe une instruction particulière, **input()**, qui attend que l'utilisateur tape quelque chose au clavier et qui prend pour valeur la chaîne de caractères correspondante. On l'utilise très souvent sous la forme **nomVariable=input()** de sorte que la variable **nomVariable** contiendra ce qui est tapé au clavier.

```

1 >>> a=input()
2 3.5

```

Précisons que **3.5** est ici ce que l'utilisateur a tapé au clavier; cette instruction, comme les autres affectations, n'affiche rien. Ensuite, on a :

```

1 >>> a
2 '3.5'

```

On voit que la valeur donnée à **a** est une chaîne de caractères; si on veut plutôt récupérer une valeur numérique, on écrira **int(input())** ou **float(input())** pour obtenir une valeur respectivement entière ou en virgule flottante.

Une variante consiste à donner une chaîne de caractères en argument à **input**; cette chaîne sera affichée à l'écran avant de lire ce que tape l'utilisateur. Cela permet par exemple de poser une question pour guider la réponse attendue.

```

1 >>> a=int(input('Combien_en_voulez-vous?'))
2 Combien en voulez-vous?42
3 >>> a
4 42

```

## II.5 Echanger le contenu de deux variables

On se place dans l'état 

|   |
|---|
| x |
| 1 |

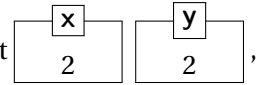
|   |
|---|
| y |
| 2 |

 et on souhaite placer le contenu de la variable **y** dans la variable **x** et le contenu de la variable **x** dans la variable **y**.

Un premier essai infructueux consiste à évaluer les instructions :

```
1 >>> x=y
2 >>> y=x
```

Cela ne fonctionne pas, car après l'évaluation de la première instruction, on se retrouve dans l'état



et la seconde instruction nous ramène dans cet état.

Lorsque l'on écrit ces deux instructions, en se trompant donc, on a une autre vision des choses : on pense en effet que l'expression  $x$  dans l'instruction  $y=x$  fait référence à la valeur initiale de  $x$  avant d'exécuter les instructions. En fait, et c'est naturel, on imagine que ces deux instructions forment un bloc et qu'elles seront exécutées en même temps. Or, ce n'est pas le cas ; les instructions sont exécutées l'une après l'autre.

La façon la plus naturelle et la plus générale pour placer dans  $y$  dans l'ancienne valeur de  $x$  consiste à la stocker dans une variable auxiliaire :

```
1 >>> t=x
2 >>> x=y
3 >>> y=t
```

On va exécuter ces instructions l'une après l'autre :

- après la première, on passe dans l'état
 

|   |   |   |
|---|---|---|
| x | y | t |
| 1 | 2 | 1 |
- après la deuxième dans l'état
 

|   |   |   |
|---|---|---|
| x | y | t |
| 2 | 2 | 1 |
- et enfin après la troisième dans l'état
 

|   |   |   |
|---|---|---|
| x | y | t |
| 2 | 1 | 1 |

On verra plus loin une solution spécifique à Python utilisant des 2-uplets.

### Entrainement 3

On considère un état dans lequel sont définies trois variables  $x$ ,  $y$  et  $z$ .

Décrire une suite d'instructions permettant de placer :

- le contenu de  $x$  dans  $z$ ,
- le contenu de  $y$  dans  $x$ ,
- le contenu de  $z$  dans  $y$ .

Une solution :

```
1 >>> x=1
2 >>> y=2
3 >>> z=3
```

```

4 >>> t=z  #Le contenu initial de z est placé dans une variable auxiliaire.
5 >>> z=x
6 >>> x=y
7 >>> y=t
8 >>> x
9 2
10 >>> y
11 3
12 >>> z
13 1

```

### III Les types composés

Sont de type composé les valeurs formées de plusieurs valeurs de types plus simples. Par exemple, les couples d'entiers sont de type composé.

#### III.1 Les $n$ -uplets

##### a Construction

Un  $n$ -uplet, *tuple* en anglais, est une généralisation du concept de couple ou de triplet. On peut voir ce type comme la traduction informatique d'un produit cartésien d'ensembles. Comme en mathématiques, pour construire une expression  $n$ -uplet, il suffit de placer des expressions entre parenthèses séparées par des virgules.

Voici un couple constitué d'un entier et d'un flottant :

```

1 >>> (1, 2.2)
2 (1, 2.2)

```

et un triplet d'entiers :

```

1 >>> (0, 1, 2)
2 (0, 1, 2)

```

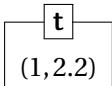
##### b Accès aux composantes

Comme pour les autres valeurs, il est possible de stocker un  $n$ -uplet dans une variable :

```

1 >>> t=(1, 2.2)

```

On arrive alors dans l'état . On remarque que, dans cette instruction, on peut omettre les parenthèses. Ainsi, l'instruction suivante lui est équivalente :

```
1 >>> t=1,2.2
```

Pour accéder aux composantes d'un  $n$ -uplet, c'est-à-dire aux sous-valeurs qu'il contient, on utilise l'expression  $t[i]$  où  $i$  est l'indice de la composante.

### ⚠ Attention danger !

➤ En Python, on commence à numéroté à partir de 0 et non de 1.

Pour obtenir la première composante du couple, on pourra alors évaluer :

```
1 >>> t[0]
2 1
```

et pour la seconde :

```
1 >>> t[1]
2 2.2
```

### ⚠ Attention danger !

⚡ Les  $n$ -uplets sont immuables. Cela signifie qu'il n'est pas possible d'affecter de nouvelles valeurs aux composantes. Ainsi l'instruction suivante produit une erreur.

```
1 >>> t[0]=5
2 TypeError: 'tuple' object does not support item assignment
```

## c Déconstruction

Il est également possible de déconstruire un  $n$ -uplet en affectant simultanément ses composantes à différentes variables.

L'instruction

```
1 >>> x,y=t
```

amène ainsi dans l'état

|         |   |     |
|---------|---|-----|
| t       | x | y   |
| (1,2.2) | 1 | 2.2 |

.

Il est important de remarquer que l'expression  $n$ -uplet est évaluée avant de faire les affectations. On peut donc s'en servir pour échanger deux variables d'un coup :

```
1 >>> x,y=y,x
```

En effet,  $y,x$  s'évalue comme le couple de première composante la valeur de  $y$  et de seconde composante la valeur de  $x$ . Une fois ce couple calculé, on effectue les affectations. C'est une façon élégante, mais spécifique à Python, de résoudre le problème de l'échange des contenus de deux variables.

**d Cas où  $n = 0$  ou 1**

Il n'existe qu'un 0-uplet, c'est celui qui contient rien, il s'écrit `()`. C'est une valeur un peu étrange dont l'utilité est limitée. Plus courant, un 1-uplet ne contenant que la valeur **v** s'écrit `(v,)`. Ici, la virgule finale est essentielle, car c'est elle qui distingue le 1-uplet `(v,)` de l'expression `(v)` dont la valeur est **v**.

**e Concaténation**

Il est possible de coller un  $n$ -uplet et un  $p$ -uplet pour obtenir un  $(n + p)$ -uplet. On parle de *concaténation*. L'opérateur correspondant en Python est l'opérateur `+`.

```
1 >>> (1,2)+(3,4,5)
2 (1, 2, 3, 4, 5)
```

**f Test d'appartenance**

Il est possible de tester si une valeur appartient à un  $n$ -uplet l'aide de l'opérateur `in`.

```
1 >>> 3 in (1,2,3)
2 True
```

**g Longueur d'un  $n$ -uplet**

On obtient la longueur d'un  $n$ -uplet à l'aide de la fonction `len` :

```
1 >>> len((1,2))
2 2
3 >>> len(())
4 0
```

**III.2 Chaînes de caractères****a Construction**

Le type des chaînes de caractères, *string* en anglais et dans Python, est celui permettant de représenter des textes. On considère dans un premier temps des textes élémentaires, ceux composés d'une unique lettre ; on les appelle les *caractères*.

En Python, les caractères peuvent être n'importe quelle lettre de l'alphabet, mais aussi des symboles, comme les signes de ponctuation :

```
1 >>> 'a'
2 'a'
3 >>> '?'
4 '?'
```

Une chaîne de caractères est suite finie de caractères consécutifs, qu'on note entre apostrophes ou guillemets :

```
1 >>> 'Ceci_est_une_chaine'
2 'Ceci_est_une_chaine'
3 >>> "Cela_en_est_une_autre"
4 'Cela_en_est_une_autre'
```

La chaîne vide se note "" ou "".

## b Accès à un caractère

Comme pour les  $n$ -uplets, on peut stocker une chaîne dans une variable :

```
1 >>> s = 'Bonjour'
```

et accéder à chacun des caractères à l'aide de la construction `s[i]` :

```
1 >>> s[2]
2 'n'
```

Comme les  $n$ -uplets, les chaînes de caractères sont immuables :

```
1 >>> s[0] = 'A'
2 TypeError: 'str' object does not support item assignment
```

## c Concaténation

Comme pour les  $n$ -uplets, on concatène deux chaînes à l'aide de l'opérateur `+` :

```
1 >>> 'Bonjour' + '_lecteur!'
2 'Bonjour_lecteur!'
```

## d Longueur

Comme pour les  $n$ -uplets, on utilise `len` pour obtenir la longueur d'une chaîne :

```
1 >>> len('Bonjour')
2 7
```

## e Sous-chaînes

Un ensemble de caractères consécutifs à l'intérieur d'une chaîne s'appelle une sous-chaîne. Ainsi, `'lecteur'` ou `'jour lec'` sont des sous-chaînes de `'Bonjour lecteur !'`. Pour extraire une sous-chaîne de `s`, on écrit `s[i:j]` où `i` est l'indice du premier caractère de la sous-chaîne et `j` est l'indice du dernier caractère **plus un**.

Par exemple :

```
1 >>> s = 'Bonjour_lecteur!'
2 >>> len(s)
3 16
4 >>> s[0:7]
5 'Bonjour'
6 >>> s[8:15]
7 'lecteur'
```

Si  $j \leq i$ , Python renvoie la chaîne vide. Si  $j$  dépasse la longueur de la chaîne, la sous-chaîne s'arrête au dernier caractère de la chaîne.

### f Test d'appartenance

Comme pour les  $n$ -uplets, l'opérateur **in** sert à tester l'appartenance d'un caractère à une chaîne.

```
1 >>> 'o' in 'Bonjour'
2 True
3 >>> 'b' in 'Bonjour'
4 False
```

On peut aussi tester la présence d'une sous-chaîne dans une chaîne avec la même construction :

```
1 >>> 'lecteur' in 'Bonjour_lecteur!'
2 True
```

### g Conversion vers des types simples

On peut convertir une valeur d'un type simple vers une chaîne de caractères à l'aide de la construction **str(e)**. La chaîne obtenue est la même que celle affichée par Python lors de l'évaluation de **e**.

```
1 >>> str(1.2)
2 '1.2'
```

Il est possible de reconvertir une telle chaîne vers une valeur d'un type simple :

```
1 >>> int('123')
2 123
3 >>> float('1.2')
4 1.2
5 >>> bool('True')
6 True
```

**Entrainement 4**

On suppose que la variable **C** est affectée d'une chaîne de caractères **C='abcde'**.

Que produisent les instructions suivantes?

```
1 >>> L=len(C)
2 >>> C[0]
3 >>> C[L]
4 >>> C[L-1]
5 >>> C[0:L-1]
6 >>> C[0:L]
7 >>> C[0:3]
8 >>> C[2:3]
9 >>> C[2:2]
```

*Testons dans la console :*

```
1 >>> C='abcde'
2 >>> L=len(C)
3 >>> L
4 5
5 >>> C[L]
6 IndexError: string index out of range
7 >>> C[L-1]
8 'e'
9 >>> C[0:L-1]
10 'abcd'
11 >>> C[0:L]
12 'abcde'
13 >>> C[0:3]
14 'abc'
15 >>> C[2:3]
16 'c'
17 >>> C[2:2]
18 ''
```



### III.3 Listes

Une *liste* est un  $n$ -uplet dont on peut changer les composantes.

Pour construire une liste, on remplace les parenthèses par des crochets :

```
1 >>> [1,2,3]
2 [1, 2, 3]
```

Toutes les opérations vues pour les  $n$ -uplets sont définies :

```
1 >>> L=[1,2]
2 >>> L[1]
3 2
4 >>> L+[3,4]
5 [1, 2, 3, 4]
```

Ici cependant, aucune erreur n'apparaît si on change la valeur d'un élément :

```
1 >>> L[0]=5486
```

En évaluant l'expression **L**, on se rend compte que cet élément a bien changé :

```
1 >>> L
2 [5486, 2]
```

### III.4 Conversions

Il est possible de convertir des  $n$ -uplets en listes et réciproquement, comme pour les types simples :

```
1 >>> list((1,2,3))
2 [1, 2, 3]
3 >>> tuple([1,2])
4 (1, 2)
```

On peut également éclater une chaîne et la convertir en la liste de ses caractères :

```
1 >>> list('Bonjour')
2 ['B', 'o', 'n', 'j', 'o', 'u', 'r']
```



#### Savoir-faire : convertir une expression d'un type à un autre

On veut parfois transformer une expression d'un type donné en une expression d'un autre type. Par exemple, si on lit une chaîne de caractères **s** dans un fichier de données et si cette chaîne représente un nombre, il se peut qu'on veuille l'interpréter comme un flottant ou un entier. Il suffit alors d'écrire **float(s)** ou **int(s)** pour demander à Python de convertir cette chaîne en flottant ou en entier.

De manière générale, les fonctions **list**, **tuple**, **str**, **float** et **int** prennent un argument et le transforment dans la mesure du possible en un objet de type liste,  $n$ -uplet, chaîne de caractères, flottant et entier respectivement.

## IV Trois exercices corrigés

### Entrainement 5

Quel est le type adapté pour représenter les données suivantes :

1. le nom d'une personne ;
2. les coordonnées d'un point mobile dans l'espace ;
3. un numéro de téléphone.

1. Une chaîne de caractères : il y a peu de chances de devoir modifier ce nom au cours du programme.
2. Si le point est amené à se déplacer, il faut pouvoir modifier ses coordonnées : on utilise une liste.
3. On pourrait être tenté d'utiliser un entier, mais le 0 en début de numéro poserait problème. Comme on n'a pas à modifier l'un des chiffres en cours de programme, on utilise une chaîne de caractères, ou un 10-uplet d'entiers, voire de caractères.

### Entrainement 6

Ecrire un script qui demande à l'utilisateur une chaîne de caractères et qui affiche le booléen **True** si les 6 voyelles de l'alphabet sont présentes dans cette chaîne, et le booléen **False** sinon

Ecrivons le script dans l'éditeur :

```
1 C=input("Une_chaine_de_caractères_svp:")
2
3 B=('a' in C) and ('e' in C) and ('i' in C) and ('o' in C) and ('u' in C) and ('y' in C)
4
5 print(B) #affichage du booléen B
```

Testons avec la console :

```
1 Une chaine de caractères svp:cryptographique
2 True
```

```
1 Une chaine de caractères svp:azerty
2 False
```

**Entrainement 7**

Ecrire un script qui demande à l'utilisateur une chaîne de caractères et qui affiche le booléen **True** si la chaîne de caractères commence par une majuscule et se termine par un point, et le booléen **False** sinon.

Le script est écrit dans l'éditeur :

```
1 C=input("Une_chaine_de_caractères_svp:")
2
3 B=(C[len(C)-1] == '.' ) and (C[0] in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
4
5 print(B)
```

et les tests sont effectués avec la console :

```
1 Une chaine de caractères svp:Essai.
2 True
```