Types de référence en Python

Les langages sur lesquels nous travaillerons cette année sont Python et SQL. Chacun de ces langages a ses propres spécificités :

- Python est un ***langage généraliste*** c'est-à-dire qu'il permet - en théorie ! - de programmer tout ce qu'il est possible de programmer ;
- SQL est un ***langage dédié*** à la consultation et à la manipulation de ***bases de données***.

Python est un langage inventé en 1989 par **Guido van Rossum**[1], gratuit et gratuitement utilisable y compris pour des utilisations commerciales. En revanche, il existe de nombreuses distributions et variantes... et on peut s'y perdre !

SQL est un langage standardisé par IBM en 1970 et libre de droits. Il est depuis lors utilisé par tous les ***Serveurs de Gestion de Bases de Données*** (***SGBD***) qu'ils soient commerciaux ou gratuits.

I. Un peu d'histoire et de théorie des langages généralistes

I.1. La préhistoire

Écrire directement du code exécutable par un processeur est une tâche... surhumaine ! C'était pourtant la seule façon de programmer les ordinateurs avant que **Grace Hopper**[2] écrive en 1950 le premier ***compilateur***.

Définition 2.1 (Compilateur)

En simplifiant un peu : un ***compilateur*** est un programme informatique permettant de transformer un ***code source*** écrit dans un langage de programmation en ***code machine***, c'est-à-dire en code directement exécutable par un processeur.

En 1959, partant du principe qu'un bon langage de programmation doit ressembler au langage que nous parlons quotidiennement, elle récidive et invente le langage ***COBOL***.

Entretemps, **John Backus**[3] invente en 1954 le langage ***FORTRAN*** pour le compte d'IBM. Il faudra deux ans de travail à son équipe pour écrire le premier compilateur FORTRAN en 1956. **Grace Hopper** travaillera elle aussi au développement du FORTRAN pour le compte d'IBM à partir de 1957.

Remarque

Aussi incroyable que cela puisse paraître, le langage FORTRAN reste très utilisé notamment parce que son ancienneté lui permet de disposer de très nombreuses ***bibliothèques de fonctions*** (notamment pour le calcul scientifique) et que sa proximité avec le code machine produit des exécutables extrêmement rapides.

Suivent alors toute une série de langages généralistes parmi lesquels, pour ne citer que les plus répandus, ... !

1. **Guido van Rossum**(1956 ;...), développeur néerlandais, a travaillé pour Google (2005-2012), travaille actuellement pour Dropbox.
2. **Grace Hopper**(1906 ;1992), mathématicienne, informaticienne et amiral de la Marine américaine.
3. **John Backus**(1924 ;2007), informaticien et théoricien des langages. Coauteur de la forme de Backus-Naur.

I.2. Différentes formes de langages généralistes

a) Bas niveau, haut niveau

Définition 2.2 (Langage de bas/haut niveau)

On dit d'un langage généraliste qu'il est ***de bas niveau*** si sa syntaxe se rapproche de celle du code machine.
On dit au contraire d'un langage généraliste qu'il est ***de haut niveau*** si sa syntaxe se rapproche de celle de la langue naturelle, essentiellement de l'anglais !

L'avantage des langages de bas niveau est qu'ils engendrent (lorsqu'ils sont bien programmés !) des exécutables extrêmement efficaces et rapides. En revanche, comme leur syntaxe est proche de celle du code machine, les codes sources écrits dans ces langages sont généralement très longs, difficiles à lire et difficiles à écrire !
Pour les langages de haut niveau, c'est le contraire ! Pour qu'ils soient proches de la langue naturelle (l'anglais), les différentes commandes du langage correspondent en fait à de nombreuses instructions machine. Ceci engendre des codes sources compacts, très lisibles (dès qu'on sait un peu parler anglais...) et faciles à écrire. En revanche, leur exécution est souvent moins efficace et rapide qu'un programme de même nature bien programmé à l'aide d'un langage de bas niveau.
Le développement de l'informatique a de façon assez naturelle conduit à attribuer aux langages de bas et de haut niveau des places et fonctions distinctes et complémentaires.
La communication directe avec les composants numériques se fait à l'aide de langages de bas niveau qui fournissent des ***bibliothèques de fonctions*** efficaces et utilisables par d'autres langages. C'est souvent le cas par exemple des pilotes de périphériques.
La construction d'applications évoluées faisant intervenir des comportements complexes et des interactions avec l'utilisateur se fait à l'aide de langages de haut niveau permettant un gain de temps pour le développement du code (compacité) et sa mise à jour (lisibilité).

b) Langage compilé/interprété

La définition 2.1 s'applique aux langages dits ***compilés***. C'est le cas de tous les langages de bas niveau.
Cependant, pour les langages de haut niveau, la compilation présente un inconvénient : en effet, il faut d'abord écrire le code, puis le compiler et enfin exécuter l'application générée par le compilateur. Ceci interdit d'obtenir de façon ***immédiate et interactive*** le résultat d'une ligne de calcul par exemple. Pour pallier cet inconvénient, une seconde démarche est possible.

Définition 2.3 (Interpréteur)

Un interpréteur est un programme capable d'exécuter lui-même un code source sans avoir à le compiler.

On parle dans ce cas de ***langage interprété***. Ce fonctionnement présente un deuxième avantage (le premier étant .....) : comme le code source est exécuté par un interpréteur capable de le comprendre, ***le code source peut se modifier lui-même en cours d'exécution du programme***. On dit que le langage est ***dynamique***.

I.3. Python

Python est un langage ***de haut niveau, interprété donc dynamique, disposant d'un très grand nombre de bibliothèques d'objets et de fonctions*** (on parle plutôt de ***modules*** pour Python). Il dispose d'une syntaxe très simple, très compacte et de types de données évolués adaptés à la

plupart des problèmes à traiter. On l'a déjà dit, c'est un langage gratuit, Open Source, disposant d'une communauté large et très active de développeurs, ce qui en fait un langage en constante progression.

Le langage Python a connu une bifurcation importante fin 2008. Une nouvelle version de Python, la version 3.0, a été publiée dont la syntaxe était incompatible avec la version 2.6 publiée simultanément. Durant plusieurs années, les deux branches de Python (2.x et 3.x) ont évolué simultanément. Aujourd'hui, le développement de la version 2.x est définitivement abandonné avec la version 2.7.5.

Python est disponible dans de nombreuses ***distributions***. Une distribution comprend un interpréteur pour Python 2.7 ou 3.x, un certain nombre de modules et éventuellement des utilitaires ou logiciels permettant par exemple de créer un exécutable à partir d'un code Python. Le choix a été fait d'installer https ://www.anaconda.com/download/ au lycée, et il vaut mieux installer cette distribution de ***Python 3*** sur votre machine.

II. Types de référence Python

II.1. Les types numériques

a) Le type `NoneType`

Il s'agit d'un type particulier, ne possédant qu'une unique valeur possible : `None`

Ce type modélise le résultat des fonctions qui ne renvoie ***aucun résultat*** : de façon intuitive, il représente la valeur ***Rien***. Sur le plan numérique, la valeur de `None` est 0. Nous reparlerons du type `NoneType` et de la valeur `None` dans le chapitre 4 sur les fonctions.

Exemple :

```
>>> None==0
False
>>> type(None)
<class 'NoneType'>
>>> a=print('Bonjour !')
Bonjour !
>>> a
```

b) Les booléens

Définition 2.4 (Booléens)

Le type ***booléen*** - ou `bool` en Python - est le type du résultat d'un test ou d'une opération logique.
Les booléens ont uniquement deux valeurs possibles : `True` et `False`.

Exemples :

```
>>> -3<2
True
>>> type(-3<2)
<class 'bool'>
```

```
>>> 1==True
True
>>> 1 is True
False
```

On rappelle les opérateurs suivants conduisant à des résultats booléens :

Opérateurs sur des expressions de type quelconque conduisant à un résultat booléen

Nom	Symbole	Exemple
$=$	`==`	`X==3`
$\neq$	`!=`	`X!=3`
$<$	`<`	`X<3`
$\leqslant$	`<=`	`X<=2.7`
$>$	`>`	`X>5`
$\geqslant$	`>=`	`X>=-2`
Identité	`is`	`1 is True`
$\in$	`in`	`1 in [0,2,4,6]`

Opérateurs sur des expressions de type booléen conduisant à un résultat booléen

Nom	Symbole	Exemple
et	`and`	`(X<=6)and(X>2)`
ou	`or`	`(X<0)or(X>=3)`
non	`not`	`not(X<=2.5)`

c) Les entiers

Définition 2.5 (Entiers)

Le type ***entier*** - `int` en Python - représente l'ensemble $\mathbb{Z}$, ***à priori sans limitation de taille*** (voir remarque ci-dessous) : pour cette raison, les calculs ne faisant intervenir que des entiers en Python sont exacts, et de précision ***théoriquement*** infinie.
Il existe deux moyens de déclarer un entier :
par son écriture décimale `n=197`
par son écriture binaire `p=0b110101`

Remarque

La seule limitation sur la taille des entiers manipulables en Python est due aux contraintes machines. Nous verrons au chapitre 3 la façon dont les entiers sont stockés en mémoire, mais on peut affirmer d'ores et déjà que la mémoire étant finie, la taille des entiers que l'on peut y stocker l'est aussi.

d) Les flottants

Définition 2.6 (Flottants)

Le type ***flottant*** - `float` en Python -, aussi appelé ***nombres à virgule flottante***, représente ***une petite partie*** de l'ensemble $\mathbb{Q}$. Nous verrons au chapitre 3 quels sont les nombres qui sont véritablement représentables comme des flottants.
Ce qu'il faut retenir d'emblée concernant leur usage : ***les calculs sur les flottants ne sont en général pas exacts*** (ils sont presque toujours approchés), ***leur précision est au mieux de 16 chiffres significatifs***.
Il existe deux moyens de déclarer un flottant :
par son écriture décimale `n=197.` ou `-2.7`
par son écriture scientifique `N=6.02e23`

e) Quelques exemples concernant les entiers et les flottants

```
>>> p=0b110101
>>> print(p)
53
>>> print(type(p))
<class 'int'>
>>> print(p-20)
33
>>> print(p**20)
30585627290848204916791848989276401
>>> print(type(1),type(1.))
<class 'int'> <class 'float'>
```

```
>>> r=0.1
>>> print(10*r)
1.0
>>> print(r+r+r+r+r+r+r+r+r+r)
0.9999999999999999
>>> import numpy as np
>>> print(np.pi)
3.141592653589793
>>> print(np.sin(2*np.pi))
-2.4492935982947064e-16
```

f) Les complexes

Définition 2.7 (Complexes)

Le type ***complexe*** - `complex` en Python - représente ***une petite partie*** de l'ensemble $\mathbb{C}$. Comme en mathématiques, ils sont constitués d'une partie réelle et d'une partie imaginaire (qui sont ***toujours deux flottants***).
Comme pour les flottants : les calculs faisant intervenir des nombres complexes sont approchés, d'une précision au mieux égale à 16 chiffres significatifs pour la partie réelle et la partie imaginaire.
Pour déclarer un nombre complexe, on le donne sous forme algébrique par la somme de sa partie réelle ***et de sa partie imaginaire multipliée par*** `1j` :

```
>>> c=1+1j
>>> x=c.real
>>> y=c.imag
>>> print(type(x),type(y))
<class 'float'> <class 'float'>
```

```
>>> z1=-3.5+3.5*1j
>>> print(z1**2)
-24.5j
>>> z2=-3.5+3.5j
>>> z1==z2
True
```

Remarque

Nous verrons au chapitre 6 qu'il est aussi possible de déclarer un nombre complexe sous sa forme exponentielle, à savoir sous la forme $re^{i\theta}$ où $r \in \mathbb{R}_+$, $\theta \in \mathbb{R}$.

g) Précisions concernant les opérations sur les types numériques

Les opérateurs suivants peuvent être utilisés avec des opérandes de ***type numérique quelconque*** :

Nom	Symbole	Exemple
Multiplication	*	a*b
Addition	+	a-b
Quotient décimal	/	a/b
Puissance	**	a**b

Les opérateurs suivants peuvent être utilisés avec des opérandes de ***type numérique à l'exception du type complexe*** :

Nom	Symbole	Exemple
Quotient euclidien	//	a//b
Reste euclidien	%	a%b

h) Transtypage et exemples

Il est possible de passer d'un type numérique à un autre : c'est ce que l'on appelle le transtypage. Pour cela, on utilise les ***fonctions de transtypage*** dont le nom est le même que celui du type.
Par ailleurs, le résultat d'une opérations sur ***deux objets de types différents*** est automatiquement transtypée. C'est aussi le cas de certaines opérations sur deux opérandes de même type, dont le résultat est naturellement d'un type différent.

Attention : on ne peut pas transtyper un complexe en flottant ou en entier. Exemples :

```
>>> int(False)
0
>>> int(True)
1
>>> True+True
2
>>> bool(-3)
True
>>> bool(0.+0.j)
False
```

```
>>> 5/7
0.7142857142857143
>>> (-1)**0.5
(6.123233995736766e-17+1j)
>>> complex(0)
0j
>>> float(-3+7j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
>>> int(-5.2)
-5
```

i) Remarques à propos des booléens

1) Le transtypage en booléen d'un objet Python (c'est-à-dire l'expression `bool(objet)` donne le résultat suivant :

   `False` si l'objet est vide, nul ou `None`

   ```
   >>> bool(0)
   False
   >>> bool(None)
   False
   ```

   `True` dans tous les autres cas

   ```
   >>> bool(1)
   True
   >>> bool(-2.5)
   True
   >>> bool(1j)
   True
   ```

2) Les deux codes ci-dessus montrent que les opérateurs `and`, `or` et `not` pourraient s'écrire à l'aide respectivement des opérations $\times$, $+$ et $1-$ ainsi que du transtypage du résultat.

   ```
   >>> b1=True;b2=False
   >>> bool(b1*b1),bool(b1*b2),bool(b2*b2)
   (True, False, False)
   >>> bool(b1+b1),bool(b1+b2),bool(b2+b2)
   (True, True, False)
   >>> bool(1-b1),bool(1-b2)
   (False, True)
   ```

   ```
   >>> b1=True;b2=False
   >>> b1 and b1,b1 and b2,b2 and b2
   (True, False, False)
   >>> b1 or b1,b1 or b2,b2 or b2
   (True, True, False)
   >>> not b1,not b2
   (False, True)
   ```

   Ce serait évidemment nettement moins lisible :-) Mais c'est une remarque qui est utile en SI pour la simplification des circuits de portes logiques.

j) Enchaînement d'opérateurs logiques

Python admet des syntaxes du type

```
>>> x=2.79
>>> 2<x<3
True
```

où l'on ***enchaîne plusieurs opérateurs de comparaison***. Le sens exact de l'exemple `2<x<3` donné est `(2<x)and(x<3)`.

Dans la plupart des cas, ***les enchaînements d'opérateurs de comparaisons sont à éviter***, sauf dans les cas simples comme l'exemple précédent. Dans les autres cas, on utilise les opérateurs booléens `and`, `or` et `not` et des parenthèses aux bons endroits !

Ex. 2.1 On dispose d'une fonction `f` prenant en paramètre un objet de type `float` et retournant un objet du même type. On suppose par ailleurs que l'on dispose de trois `float` `a`, `b` et `c` vérifiant `a<b<c`.
1) Donner des exemples de valeurs de `f(a)`, `f(b)` et `f(c)` pour lesquelles `((f(a)>f(b))and(f(b)<f(c)))or((f(a)<f(b))and(f(b)>f(c)))` vaut `True`.
2) Une fonction qui vérifie de telles conditions peut-elle être monotone ?

Cor. 2.1

II.2. Séquences

La notion de ***séquence*** regroupe en Python plusieurs types ayant tous la caractéristique d'être des « collections ordonnées d'objets » : on y trouve essentiellement ***les listes (`list`), les t-uplets (`tuple`) et les chaînes de caractères (`str`)***. Mais il y en a d'autres.

a) Définition

- Une ***liste*** (`list`) est une ***suite ordonnée d'objets***, éventuellement de types différents, avec répétition possible. Syntaxe générale de la déclaration d'une liste
  `L=[l0,l1,l2,...]`
- Un ***t-uplet*** (`tuple`) est une ***suite ordonnée d'objets***, éventuellement de types différents, avec répétition possible. Syntaxe générale de la déclaration d'un t-uplet
  `T=t0,t1,t2,...` ou bien `T=(t0,t1,t2,...)`

Remarque

Les types `list` et `tuple` sont en apparence redondants. Nous verrons à la sous-section f) qu'ils ont bien des rôles différents.

- Une ***chaîne de caractères*** (`str`) est une ***suite ordonnée de caractères alphanumériques*** : chiffres, lettres, et à peu près tous les symboles imaginables. Syntaxe générale de la déclaration d'une chaîne
  `C="c0c1c2..."` ou bien `C='c0c1c2...'` où `c0`, `c1`... sont des caractères.
  Comme on le voit une chaîne de caractères peut être délimitée ***soit par deux guillemets***, ***soit par deux apostrophes*** (soit... voir au chapitre 4 !). C'est une façon simple de permettre d'insérer un guillemet ou une apostrophe dans une chaîne :

```
>>> s1="C'est un str avec une apostrophe incluse.";print(s1)
C'est un str avec une apostrophe incluse.
>>> s2='Voici au contraire un str avec deux "guillemets" inclus.';print(s2)
Voici au contraire un str avec deux "guillemets" inclus.
```

  Une autre façon d'obtenir guillemet ou apostrophe dans une chaîne est ***de les faire précéder par un*** \ (*backslash*). On dit que \ est un ***caractère d'échappement*** qui permet d'obtenir certains caractères spéciaux.

```
>>> s3="C'est un str avec une apostrophe et deux \"guillemets\" inclus.";print(s3)
C'est un str avec une apostrophe et deux "guillemets" inclus.
```

Remarque

`[]` est une liste vide, `()` un t-uplet vide, `""` (ou `''`) une chaîne de caractères vide.

```
>>> [],type([])
([], <class 'list'>)
>>> (),type(())
((), <class 'tuple'>)
```

```
>>> "",type("")
('', <class 'str'>)
>>> '',type('')
('', <class 'str'>)
```

Voici quelques caractères spéciaux fréquemment rencontrés dans une chaîne de caractères :

Caractères	Effet	Caractères	Effet	Caractères	Effet
\n	Retour à la ligne	\t	Tabulation	\b	Backspace (retour d'un caractère en arrière)
\'	Apostrophe	\"	Guillemet	\\	Permet de faire un « vrai » backslash

b) Transtypage

Comme dans le cas des types numériques, on peut passer d'un type séquence à un autre à l'aide d'un transtypage. Les fonctions de transtypage sont en l'occurrence `list(seq)`, `tuple(seq)` et `str(seq)`.

```
>>> s="abc123";s
'abc123'
>>> l=list(s);l
['a', 'b', 'c', '1', '2', '3']
>>> t=tuple(l);t
('a', 'b', 'c', '1', '2', '3')
>>> S=str(t);S
"('a', 'b', 'c', '1', '2', '3')"
>>> S==s
False
>>> s2=''.join(l);S
"('a', 'b', 'c', '1', '2', '3')"
>>> s2==s
True
```

Comme on le voit sur cet exemple, l'effet de chaque fonction correspond à ce à quoi on s'attend intuitivement, à l'exception de `str(seq)` dont l'effet est de créer une chaîne de caractères représentant la valeur du paramètre.

Remarque

`str(objet)` renvoie la représentation par défaut de l'objet (c'est-à-dire celle affichée dans la console).
La fonction `eval` permet d'obtenir à nouveau l'objet qui avait été transtypé en chaîne de caractères.

```
>>> S=str(1+3j)
```

```
>>> eval(S);type(eval(S))
(1+3j)
<class 'complex'>
```

c) Opérations sur les séquences

Définition 2.8 (Concaténation)

On peut « ajouter » ***deux séquences*** `a` ***et*** `b` ***de même type*** ..... à l'aide de l'opération `a+b`.
On appelle ***concaténation*** cette opération qui consiste en fait à former une séquence du même type que `a` et `b` en mettant à la suite des éléments de `a` ceux de `b`.

On peut aussi ***multiplier une séquence*** `s` ***par un entier*** `n` à l'aide de l'opération `n*s` ou `s*n` ce qui a pour effet :

- de donner une séquence de même type ***vide*** si l'entier est négatif ou nul ;
- de former la séquence $\overbrace{\texttt{s}+\texttt{s}+\ldots+\texttt{s}}^{\texttt{n fois}}$ si l'entier est strictement positif.

```
>>> a=[1,2,5,9];b=[1,3,5,7]
>>> a+b
[1, 2, 5, 9, 1, 3, 5, 7]
```

```
>>> a*3
[1, 2, 5, 9, 1, 2, 5, 9, 1, 2, 5, 9]
>>> -1*a
[]
```

d) Slicing

Définition 2.9 (Slicing)

Le ***slicing*** (qu'on pourrait traduire par tranchage en français, mais le terme n'est pas utilisé) consiste à extraire une sous-séquence d'une séquence donnée. Plusieurs syntaxes très simples permettent d'effectuer du slicing en Python. Étant donnée une séquence `seq` :
`seq[n]` désigne le terme d'indice `n` de la séquence (voir ci-dessous)
`seq[a:b]` désigne la séquence formée des termes dont l'indice est compris entre `a` (***inclus***) et `b` (***exclus***)
`seq[:b]` désigne la séquence formée des termes dont l'indice est ***strictement inférieur*** à `b`
`seq[a:]` désigne la séquence formée des termes dont l'indice est ***supérieur ou égal*** à `a`
`seq[a:b:c]` désigne la séquence formée des termes dont l'indice va de `a` (***inclus***) à `b` (***exclus***) en augmentant par pas de `c`

Important ! Valeurs possibles des indices

- Les indices sont évidemment de type `int` ou sont éventuellement absents.
- En Python, l'indice du premier objet d'une séquence est ***0***.
- Pour une séquence formée de n objets, la valeur d'un indice i peut être comprise entre $-n$ et $n-1$:
  - ⋆ si $i \geqslant 0$, il désigne l'indice ***en partant du début de la séquence*** ;
  - ⋆ si $i < 0$, il désigne l'indice ***en partant de la fin de la séquence***, ***-1*** étant l'indice du dernier objet, etc...

Quelques exemples s'imposent !

```
>>> s3[0]
'C'
>>> s3[1]
"'"
>>> s3[41:53]
'"guillemets"'
```

```
>>> s3[:41]
"C'est un str avec une apostrophe et deux "
>>> s3[53:]
' inclus.'
>>> s3[-1::-1]
'.sulcni "stemelliug" xued te ehportsopa enu ceva rts nu tse\'C'
```

Ex. 2.2 Qu'effectue la ligne `s3[-1::-1]` ?

Cor. 2.2

Ex. 2.3 Comment ne prendre que les lettres d'indice pair/impair de la chaîne `s3` ?

Cor. 2.3

e) Divers

Le tableau suivant expose quelques fonctions ou opérateurs intéressants valables pour les séquences :

Utilité	Syntaxe	Exemple
Intervalle entier (du type $[\![a;b[\![$)	`range(a,b,c)` où `b` et `c` sont optionnels.	`>>> T=tuple(range(5,-5,-2));T` `(5, 3, 1, -1, -3)`

Somme	`sum(seq,v)` où `v` est une valeur (optionnelle) ajoutée à la somme. `seq` doit être formée d'objets de types ***numériques*** uniquement.	`>>> sum(T)`
		`5`
		`>>> sum(T,-5)`
		`0`

Longueur (nombre d'éléments)	`len(seq)`	`>>> len(s3)` `61`
Minimum	`min(seq)`	`>>> min(T)` `-3`
Maximum	`max(seq)`	`>>> max(s3)` `'x'`
`in`	`obj in seq`	`>>> "'" in s3` `True`
`not in`	`obj not in seq`	`>>> -5 not in T` `True`

f) Cas particulier des listes

Les syntaxes suivantes ***ne sont valables que pour les listes*** ! Elles déclenchent une erreur si on essaye de les appliquer aux t-uplets ou aux chaînes de caractères.

- Modification d'éléments d'une liste :
  Nous avons vu (voir sous-section d)) que l'on peut accéder à des sous-listes d'une liste donnée grâce au slicing.
  On peut en utilisant cette syntaxe modifier certains objets de la liste.

```
>>> l=list(range(10));l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[1]=9;l
[0, 9, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[3::2]=range(7,0,-2);l
[0, 9, 2, 7, 4, 5, 6, 3, 8, 1]
```

Ex. 2.4 Que vaut `list(range(7,0,-2))` ?

Cor. 2.4

Si la liste à affecter (celle du membre de gauche) n'est pas de même longueur que la liste des valeurs données dans le membre de droite, la liste à affecter est allongée ou réduite en conséquence :

```
>>> l[1:-1]='coupe';l
[0, 'c', 'o', 'u', 'p', 'e', 1]
>>> l[1:-1]=['coupe'];l
[0, 'coupe', 1]
```

Ex. 2.5 Sur l'exemple de gauche, pourquoi les deux lignes ne donnent pas le même résultat ?

Cor. 2.5

- Compréhension de liste :

Définition 2.10 (Compréhension de liste)

On appelle ***compréhension de liste*** ou ***liste définie en compréhension*** une syntaxe particulière permettant de définir les listes de la même façon que l'on définit en mathématiques ***les ensembles en compréhension***. Plus précisément, étant données une séquence `seq`, une fonction `f` et une fonction à valeurs booléennes `cond`, on a les syntaxes suivantes :

```
[f(k) for k in seq]
[f(k) for k in seq if cond(k)]
```

À nouveau, quelques exemples seront plus parlant !

```
>>> [k**2 for k in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [chr(k) for k in range(97,113)]
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p']
>>> L=[k-96 for k in range(97,123) if chr(k) in 'aeiouy'];L
[1, 5, 9, 15, 21, 25]
```

Ex. 2.6 D'après vous, que fait la fonction `chr` ?

Cor. 2.6

Ex. 2.7 Comment obtenir la liste des consonnes de l'alphabet ?

Cor. 2.7

Ex. 2.8 Comment obtenir la liste des entiers de l'intervalle $[\![0;1000]\!]$ qui sont à la fois des carrés et des cubes ?

Cor. 2.8

Ex. 2.9 La fonction `sum(seq)` renvoie la somme des éléments d'une séquence (liste ou tuplet) ***s'ils sont tous de types numériques***.

Écrire une fonction `approx_e(n)` renvoyant la somme $\sum_{k=0}^{n} \frac{1}{k!}$.

Tester votre fonction avec $n = 5$, $n = 10$, $n = 15$, comparer avec la valeur de $e = \exp(1)$.
Remarque : on pourra commencer par programmer la fonction `fact(n)` renvoyant la valeur de $n!$.

III. Quelques précisions

III.1. Retour sur l'opérateur d'affectation

- Affectations multiples : on peut affecter ***simultanément des valeurs à plusieurs variables*** :

```
>>> x,y,z=1,2,3
>>> print('x vaut ',x,', y vaut ',y,' et z vaut ',z,sep='')
x vaut 1, y vaut 2 et z vaut 3
```

- Enchaînement d'affectations : on peut aussi affecter ***simultanément la même valeur à plusieurs variables*** en enchainant les opérateurs d'affectation :

```
>>> x=y=z=1
>>> print('x vaut ',x,', y vaut ',y,' et z vaut ',z,sep='')
x vaut 1, y vaut 1 et z vaut 1
```

- Augmentation, réduction, dilatation, contraction : enfin il existe quatre autres opérateurs d'affectation qui sont `+=`, `-=`, `*=` et `/=`. Exemple :

```
>>> x+=1    # identique a x=x+1
>>> print(x)
2
>>> x-=0.5  # identique a x=x-0.5
>>> print(x)
1.5
>>> x*=2    # revient au meme que x=x*2
>>> print(x)
3.0
>>> x/=0.5  # revient au meme que x=x/0.5
```

Ex. 2.10 Quelle est la nouvelle valeur de `x` après la dernière ligne de cet exemple ?

Cor. 2.10

III.2. Retour sur le fonctionnement des variables Python

Nous verrons en détail au chapitre 3 la représentation interne des objets en Python. Pour l'instant, il faut ***retenir*** que ***ce que contient réellement une variable, c'est une adresse mémoire*** vers l'endroit de la mémoire où est stockée sa valeur. Par exemple :

```
>>> ma_variable=-2
>>> a=ma_variable
>>> b=7
```

Dans les cas simples, ce fonctionnement ne pose pas de problème.

```
>>> a=1;b=a;a=2
```

Le code précédent par exemple accomplit les tâches suivantes :

- .....
- .....
- .....
- .....
- .....
- à l'issue de ce code la valeur de `a` est donc . et la valeur de `b` est ..

Mais ça devient un peu plus subtil - ou compliqué suivant le point de vue ! - dans l'usage par exemple des listes. Que fait par exemple le code suivant ?

```
>>> a=[1,2];b=a;a[0]=3
```

- .....
- .....

- .....
- .....

Quelles sont alors les valeurs de `a` et de `b` à l'issue de ce code ?

.....

Une façon de contourner ce problème est d'utiliser le slicing :

```
>>> a=[1,2]
>>> b=a[:] # cree une nouvelle liste avec les memes elements que a
>>> a[0]=3 # on ne modifie pas b qui pointe vers une autre liste
>>> a,b
([3, 2], [1, 2])
```

III.3. Script et bloc de commandes

Définition 2.11 (Script et bloc de commandes)

Un ***bloc de commandes*** est un ensemble de commandes qui se suivent. Cette notion ne prend vraiment de sens que lorsqu'on travaille en dehors d'une console, directement sur un ***programme*** ou ***script*** Python, c'est-à-dire un fichier texte, dont l'extension est généralement `.py` et qui regroupe plusieurs lignes de codes (parfois des centaines ou des milliers !). Il existe deux manières de ***séparer deux commandes consécutives d'un bloc de commandes*** en Python

- le retour à la ligne : on entre une ligne de commande dans le script, on retourne à la ligne, on en écrit une autre etc...
- le point-virgule (`;`)
  On peut écrire plusieurs commandes sur la même ligne à condition de les séparer par un point-virgule. C'est, sauf dans des cas très simples, une mauvaise pratique : il vaut mieux que le code soit ***aéré*** afin d'être ***lisible***.

III.4. La fonction `print`

Lorsqu'on affecte une valeur à une variable, la console n'affiche rien : une affectation ne retourne aucun résultat, la variable est affectée et c'est tout ! Souvent, lorsqu'on veut vérifier la valeur d'une variable, il suffit d'écrire cette variable dans la console. J'ai ainsi souvent été amené à écrire des lignes semblables à celle-ci :

```
>>> l=list(range(3,7));l
[3, 4, 5, 6]
```

C'est une méthode rapide et tentante, mais un peu limitée. ***Notamment, elle ne fonctionne pas si le code est exécuté depuis l'éditeur de Spyder***.
Pour y remédier, il existe une fonction entièrement dédiée à l'affichage du texte ou des valeurs d'objets : la fonction `print` dont la syntaxe est la suivante
`print(obj1,obj2,obj3,...)`
Elle admet un nombre quelconque de paramètres d'entrée. Son effet est d'écrire les paramètres d'entrée sur la sortie standard (c'est-à-dire dans la console). Un exemple :

```
>>> print(l)
[3, 4, 5, 6]
```

affiche la liste `l` que nous avions définie.
Ex. 2.11 Qu'affichent les commandes `print('a\bb ')`, `print('a\nb ')` et `print('a\\b')` ?

Cor. 2.11

III.5. Python : un langage dynamique !

Pour terminer, introduisons trois nouvelles fonctions :

- `input` dont la syntaxe est `a=input(question)`
  L'effet de la fonction `input` est
  - ⋆ d'écrire la chaine de caractères `question` dans la console ;
  - ⋆ de ***lire une ligne de l'entrée standard***. En pratique, cela signifie que `input` attend que l'utilisateur écrive une chaîne de caractères puis, lorsque l'utilisateur appuie sur la touche *Entrée*, retourne cette chaîne comme résultat.

  La commande `a=input()` aura donc pour effet d'affecter à `a` la chaîne de caractères écrite au clavier par l'utilisateur.
- La fonction `eval(s)` que nous avons déjà rencontrée permet d'évaluer une expression Python donnée au moyen d'une chaîne de caractères `s` et retourne la valeur de cette évaluation. Certaines choses sont interdites dans l'expression `s`, par exemple les affectations.
- La fonction `exec` autorise quant à elle les affectations mais ne retourne rien.

La fonction `input` associée aux fonctions `eval` et `exec` font de Python un langage dynamique : on peut modifier le code à exécuter alors même que le programme est en cours d'exécution !

```
commandes=input()
exec(commandes)
```

IV. Résumé

IV.1. Types de référence

Nom	En français	Définition (informelle)	Exemple
Types numériques			

`int`	Entier	Permet de représenter des entiers relatifs aussi grands ou petits soient-ils.	`>>> a=-167;a,type(a)`
			`(-167, <class 'int'>)`
			`>>> b=0b110100010;b,type(b)`
			`(418, <class 'int'>)`

`float`	Nombre à virgule flottante	Ce sont des approximations de nombres réels représentées à l'aide de trois entiers : voir Chapitre 1, section II.6.. Ils sont limités en taille et en précision.	`>>> x=-167.;x,type(x)`
			`(-167.0, <class 'float'>)`
			`>>> y=1e-3;y,type(y)`
			`(0.001, <class 'float'>)`

`bool`	Booléen	Résultat d'un test logique. N'a que deux valeurs possibles : `True` ou `False`.	`>>> b1=(a==x);b1,type(b1)`
			`(True, <class 'bool'>)`
			`>>> b2=(a is x);b2,type(b2)`
			`(False, <class 'bool'>)`

`complex`	Complexe	L'équivalent des nombres complexes, mais avec les mêmes limitations que le type `float`.	`>>> c=5j;c**2,type(c**2)`
			`((-25+0j), <class 'complex'>)`
			`>>> z=x+y*1j;z,type(z)`
			`((-167+0.001j), <class 'complex'>)`

Séquences			
`list`	Liste	Suite ordonnée d'objets, éventuellement de types différents, avec répétition possible.	`>>> L=[a,b1,c];L,type(L)` `([-167, True, 5j], <class 'list'>)`
`tuple`	t-uplet	Suite ordonnée d'objets, éventuellement de types différents, avec répétition possible.	`>>> T=tuple(L);T,type(T)` `((-167, True, 5j), <class 'tuple'>)`

`str`	Chaîne de caractères	Suite ordonnée de caractères alphanumériques.	`>>> s="a\tb\nc\\d\'e\"f";print(s)`
			`a b`
			`c\d'e"f`
			`>>> type(s)`
			`<class 'str'>`

IV.2. Fonctions vues jusqu'ici

Dans la syntaxe donnée des fonctions suivantes, on prend la convention habituelle de mettre entre crochets `[,...]` les paramètres d'entrée facultatifs appelés ***paramètres optionnels***.

Fonction	Syntaxe	Exemple
La plus importante		

`help`	`help(fonc)`
	affiche l'aide de la fonction `fonc`.
		`>>> help(bin)`
		`Help on built-in function bin in module builtins:`
		
		`bin(number, /)`
		`Return the binary representation of an integer.`
		
		`>>> bin(2796202)`
		`'0b1010101010101010101010'`

Transtypage		
`bin`	`bin(ent)` retourne un objet de type `str` donnant l'écriture binaire d'un entier.	`>>> bin(1023)` `'0b1111111111'`
`bool`	`bool(obj)` retourne `False` si `obj` vaut 0 ou est vide, `True` sinon.	`>>> bool()` `False`
`complex`	`complex(num)` convertit en nombre complexe les autres types numériques.	`>>> complex(True)` `(1+0j)`
`float`	`float(num)` convertit en nombre réel `num` s'il est de type numérique ***non complexe***.	`>>> float(12)` `12.0`
`int`	`int(num)` convertit `num` (non complexe) en entier en supprimant la partie décimale s'il s'agit d'un `float`.	`>>> int(1.5),int(-1.5)` `(1, -1)`

Fonction	Syntaxe	Exemple

`list`	`list(seq)` transtype la séquence `seq` en une liste. Fonctionne aussi si `seq` est un ensemble.	`>>> s='abracadabra';l1=list(s)`
		`>>> l1`
		`['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']`

`str`	`str(obj)` retourne la chaîne de caractères qui serait affichée par la console pour représenter `obj`.	`>>> str(l1)` `"['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']"`
`tuple`	`tuple(seq)` transtype la séquence `seq` en un t-uplet. Fonctionne aussi si `seq` est un ensemble.	`>>> tuple(l1)` `('a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a')`
Entrée/sortie		

`format`	`format(reel,'.nf')` permet d'afficher le nombre `reel` de type `float` avec n décimale. Voir `help(format)` pour plus de détail.	`>>> format(0.1,'.25f')`
		`'0.1000000000000000055511151'`
		`>>> format(17,'04d')`
		`'0017'`

`input`	`input()` retourne la ***chaîne de caractères*** tapée par l'utilisateur.	`a=input()` attend que l'utilisateur tape une ***chaîne de caractères*** validée par la touche *Entrée* puis affecte cette ***chaîne*** à la variable `a`.

`print`	`print(a,...[,sep=s])` affiche les objets `a`, etc... séparés par la chaîne de caractère `s`.	`>>> print(chr(97),chr(98),sep='\n')`
		`a`
		`b`

Système		

`del`	`del(var)` supprime la variable `var`.	`>>> l1`
		`['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']`
		`>>> del(l1)`
		`>>> l1`
		`Traceback (most recent call last):`
		`  File "<stdin>", line 1, in <module>`
		`NameError: name 'l1' is not defined`

Fonction	Syntaxe	Exemple

eval	`eval(s)`
	évalue la chaîne `s` comme s'il s'agissait d'un code Python et retourne le résultat de cette évaluation. Les affectations sont interdites dans la chaîne `s`.
		`>>> eval('[1,2,3]')`
		`[1, 2, 3]`
		`>>> eval('l2=[1,2,3]')`
		`File "<string>", line 1`
		`l2=[1,2,3]`
		`^`
		`SyntaxError: invalid syntax`

exec	`exec(s)` comme `eval` mais les affectations sont autorisées et la fonction ne retourne rien.	`>>> exec('l2=[1,2,3];print(l2)')` `[1, 2, 3]`

id	`id(obj)`
	retourne l'adresse mémoire où est stockée l'objet `obj` ou bien l'adresse où est stockée la valeur référencée par `obj` si c'est une variable.
		`>>> id(l2)`
		`2649212723080`
		`>>> id([1,2,3])`
		`2649212071176`

type	`type(obj)`
	retourne le type d'un objet.
		`>>> type(l2)`
		`<class 'list'>`
		`>>> type(type(l2))`
		`<class 'type'>`

Divers		
abs	`abs(num)` retourne la valeur absolue ou module d'un type numérique.	`>>> abs(1+1j)` `1.4142135623730951`
chr	`chr(ent)` retourne le caractère associé à l'entier `ent` dans le standard Unicode.	`>>> chr(32)` `' '`
len	`len(seq)` retourne le nombre d'éléments d'une séquence ou le cardinal d'un ensemble.	`>>> len(list(range(5,15)))` `10`

Fonction	Syntaxe	Exemple
ord	`ord(c)` retourne l'entier associé au caractère `c` (c'est-à-dire que `c` est une chaîne avec un seul élément) dans le standard Unicode.	`>>> ord('A'),ord('@'),ord('\n')` `(65, 64, 10)`

range	`range(a[,b[,c]])`
	retourne un intervalle d'entiers avec une syntaxe proche de celle du slicing.
		`>>> L=list(range(10,5,-1))`
		`>>> print(L)`
		`[10, 9, 8, 7, 6]`

sum	`sum(seq)` retourne la somme des éléments d'une séquence s'ils sont de type numérique.	`>>> sum(L)` `40`

IV.3. Opérateurs vus jusqu'ici

Tableaux de synthèse des différents opérateurs vus jusqu'ici et de leur effet.
On désigne par `num` un type numérique, `seq` un type séquence.

Opér.	Type a	Type b	Effet
a+b	num	num	Somme
a+b	seq	seq	Concaténation
a-b	num	num	Différence
a*b	num	num	Produit
a*b	int	seq	Concaténation multiple
a/b	num	num	Division

Opér.	Type a	Type b	Effet
a//b	float ou int	float ou int	Quotient « euclidien »
a%b	float ou int	float ou int	Reste « euclidien »
a**b	num	int	Puissance
a**b	float>0	float	Puissance

- `a=b` : crée `b` s'il n'existe pas et fait pointer `a` sur l'adresse mémoire de la valeur de `b`.
- `a+=b` : signifie `a=a+b`
- `a-=b` : signifie `a=a-b`
- `a*=b` : signifie `a=a*b`
- `a/=b` : signifie `a=a/b`

Opérateur	Typ. a	Typ. b	Effet
a in b	obj	seq	Appartenance
a not in b	obj	seq	Non appartenance
a<b	float	float	Comparaison stricte
a<=b	num	num	Comparaison large
a==b	obj	obj	Égalité des valeurs
a!=b	obj	obj	Non égalité des valeurs

a is b	obj	obj	Égalité des références mémoire
a is not b	obj	obj	Non égalité des références mémoire
a and b	bool	bool	***ET*** logique
a or b	bool	bool	***OU*** logique
not a	bool		Négation logique
a!=b	bool	bool	***OU*** exclusif aussi appelé XOR