Chapitre 2

Introduction à Python et à son environnement

1. Écosystème et environnement de travail

Python est un langage de programmation polyvalent et modulaire, utilisé dans de très nombreux domaines, scientifiques ou non. C'est pourquoi il existe de nombreuses distributions de python. Au lycée Louis-le-Grand nous utilisons Pyzo, distribution gratuite et open-source livrée avec un environnement de développement intégré simple à utiliser et avec tous les modules scientifiques dont nous aurons besoin cette année.

1.1 Installation du logiciel

La première chose à faire est de télécharger Pyzo à l'adresse suivante : http://www.pyzo.org/ et de l'installer. Pyzo est disponible pour Windows, Linux et OSX (a priori si votre ordinateur n'est pas trop ancien vous aurez besoin de la version 64 bits). Comme souvent, il est conseillé d'utiliser la dernière version disponible et de vérifier de temps à autre si une nouvelle version n'est pas proposée au téléchargement.

1.2 L'environnement de travail

L'environnement de développement que vous venez d'installer est un ensemble d'outils pour programmeurs conçus pour être utilisés au sein d'un éditeur interactif nommé IEP (*Interactive Editor for Python*). IEP consiste avant tout en un éditeur et une interface système (un *shell*) ; les autres outils ne sont pas indispensables à nos besoins pour l'instant très modestes.

• L'interprète de commande

Souvenons-nous que Python est un langage interprété (voir le chapitre précédent) : il est nécessaire de disposer d'un interprète de commande pour traduire nos instructions Python en langage machine. Sous Pyzo, l'interprète de commande se trouve dans le *Shell*. Si vous utilisez la dernière version de Pyzo deux interprètes de commandes sont disponibles, et en fonction des réglages des préférences l'un des deux est lancé automatiquement au démarrage.

– S'il s'agit de l'interprète de commande par défaut le shell ressemblera à ceci :

```
Python 3.4.2 |Continuum Analytics, Inc.| (default, Oct 21 2014, 17:42:20)
on darwin (64 bits).
This is the IEP interpreter with integrated event loop for TK.
Type 'help' for help, type '?' for a list of *magic* commands.

>>>
```

– S'il s'agit de l'interprète IPython, le shell aura cette forme :

```
Python 3.4.2 |Continuum Analytics, Inc.| (default, Oct 21 2014, 17:42:20)
on darwin (64 bits).
This is the IEP interpreter with integrated event loop for PYSIDE.

Using IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

IPython est un interprète de commande amélioré offrant plus de fonctionnalités que l'interprète par défaut, mais pour débuter n'importe lequel fera l'affaire.

Quel que soit l'interprète utilisé, ce dernier se propose d'engager avec vous un dialogue interactif en vous invitant à taper une première commande après le *prompt* `>>>` ou `In [1]:`
Répondons à l'invite en tapant "1 + 1" suivi de la touche "entrée" et observons sa réponse :

```
>>> 1 + 1
2

>>>
```

```
In [1]: 1 + 1
Out[1]: 2

In [2]:
```

L'interprète de commande a exécuté notre instruction et retourné le résultat de celle-ci. Une fois la tâche terminée, le prompt vous invite de nouveau à taper une instruction. Nous pouvons déjà observer que IPython numérote les instructions et distingue clairement votre solicitation (`In`) de sa réponse (`Out`). Poursuivons le dialogue avec :

```
>>> print('Hello world !')
Hello world !

>>>
```

```
In [2]: print('Hello world !')
Hello world !

In [3]:
```

L'interprète IPython nous permet ici de discerner une différence qui serait passée inaperçue avec l'interprète de commande traditionnel. *Toute* fonction Python retourne un résultat (affiché derrière le `Out`), et *certaines* d'entres-elles ont en plus un effet sur l'environnement [1] :

- l'instruction `1 + 1` a comme résultat `2` et n'a pas d'effet sur l'environnement ;
- l'instruction `print('Hello world !')` retourne une valeur particulière nommée `None` et a pour effet d'afficher une chaîne de caractères dans le shell.

La valeur `None` est la valeur retournée par les instructions qui n'ont en quelque sorte « rien à renvoyer » ; l'interprète IPython ne s'y trompe pas et n'affiche pas cette réponse, alors qu'en toute logique on pourrait s'attendre à voir écrit dans le shell : `Out[2]: None`

La chaîne `Hello world !` qui apparaît dans le shell *n'est pas* le résultat de l'instruction `print` mais l'*effet* de celle-ci sur son environnement.
Notons que l'interprète standard ne s'y trompe pas non plus : le résultat de l'instruction (`None`, rappelons-le) n'apparaît pas dans le shell, mais avec cet interprète il n'est pas possible visuellement de distinguer une réponse d'un effet.

Quelle différence entre retour et effet sur l'environnement ?
Pour tenter de mieux comprendre celle-ci, jetons un coup d'œil sur l'aide en ligne de la fonction `print` :

```
In [3]: help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

In [4]:
```

Cette fonction possède quatre paramètres optionnels (`file`, `sep`, `end` et `flush`) qui possèdent des valeurs par défaut, valeurs utilisées lorsqu'on ne précise pas explicitement les valeurs que l'on souhaite attribuer à ces paramètres. Le premier de ceux-ci, `file`, désigne le « lieu » vers lequel doit être dirigé le flux de caractères à imprimer ; sa valeur par défaut est `sys.stdout` (pour *system standard output*) qui désigne la sortie standard, à savoir l'écran pour la majorité des systèmes, et plus précisément ici le shell. Pour s'en convaincre, modifions cette sortie en écrivant :

1. On appelle *effet* une interaction avec un périphérique de sortie ou une modification de la mémoire.

```
In [4]: print('Hello world !', file=open('essai.txt', 'w'))

In [5]:
```

Comme on peut le constater, cette fois il n'y a plus d'affichage dans le shell ; en revanche si vous tentez l'expérience vous constaterez qu'a été créé dans votre répertoire par défaut un nouveau fichier texte nommé `essai.txt` qui contient la chaîne de caractères `'Hello world !'`.

Exercice 1 Pour vérifier que vous avez bien compris cette différence entre résultat et effet, compléter l'état du shell après chacune des solicitations suivantes :

```
In [5]: 1 + 2


In [6]:
```

```
In [5]: print(1 + 2)


In [6]:
```

```
In [5]: print(print(1 + 2))


In [6]:
```

```
In [5]: print(1) + 2


In [6]:
```

```
In [5]: print(1) + print(2)


In [6]:
```

• L'éditeur de texte

Ce mode interactif est idéal pour de courtes commandes mais ne convient pas dès lors que nous aurons à rédiger des programmes. Dans ce cas, on utilise l'éditeur de texte fourni par IEP. Il possède nombre de fonctionnalités des éditeurs de texte usuels mais est doté en plus de certaines fonctionnalités propres à la programmation (numérotation des lignes, coloration syntaxique, indentation automatique, etc.). En contrepartie, pour être utilisables par IEP vos fichiers texte devront respecter certaines contraintes : leurs noms devront se terminer par le suffixe `.py`, leur contenu devra suivre les règles d'indentation propres à Python (on en reparlera plus loin), ne comporter que des instructions Python (si vous souhaitez ajouter une ligne de commentaires celle-ci devra débuter par le caractère `#`), etc.
Recopions donc les lignes suivantes dans l'éditeur :

```
print('Un nouveau calcul')
print('5 * 3 =', 5*3)
2 + 4
```

puis exécutons ce script (menu "Exécuter → Exécuter le fichier" ou mieux par l'intermédiaire du raccourci clavier). Observons se qui se passe dans le panneau de l'interprète IPython :

```
In [6]: (executing lines 1 to 3 of "<tmp 1>")
Un nouveau calcul
5 * 3 = 15

In [7]:
```

Il y a là une différence importante avec l'utilisation directe de l'interprète : lorsqu'on exécute un script écrit dans l'éditeur le résultat de ce dernier n'est pas retourné. Les effets des deux instructions `print` ont bien eu lieu, mais le résultat du calcul `2 + 4` *n'est pas visible*, alors que si ce calcul avait été exécuté directement dans le shell, nous aurions vu apparaître `Out[6]: 6`

Revenons dans le panneau d'édition et modifions son contenu pour le remplacer par le contenu suivant :

```
# calcul du nombre de secondes dans une journée
print('une journée a une durée égale à', 60 * 60 * 24, 'secondes')
```

puis exécutons ce script. Voici ce que l'on obtient :

```
In [7]: (executing lines 1 to 2 of "<tmp 1>")
une journée a une durée égale à 86400 secondes

In [8]:
```

La première ligne du script n'a pas d'effet ; comme la majorité des langages de script, les commentaires Python débutent caractère `#`. Qu'il soit utilisé comme premier caractère ou non, le `#` introduit un commentaire jusqu'à la fin de la ligne. Il ne faut jamais hésiter à commenter un code (sans pour autant tomber dans un verbiage creux) pour en faciliter la compréhension future.
Quant à la seconde ligne du script, elle vous montre que la fonction `print` peut posséder plusieurs arguments.

Exercice 1 Relire l'aide dédiée à la fonction `print` (page 2.2 de ce document) pour comprendre le rôle des paramètres optionnels `sep` et `end`, puis deviner le résultat de l'exécution du script suivant :

```
print(1, 2, 3, sep='+', end='=')
print(6, 5, 4, sep='\n', end='*')
```

• Pour résumer

Python peut être utilisé aussi bien en mode interactif qu'en mode script. Dans le premier cas, il y a un dialogue entre l'utilisateur et l'interprète : les commandes entrées par l'utilisateur sont évaluées au fur et à mesure. Cette première solution est pratique pour réaliser des prototypes, ainsi que pour tester tout ou partie d'un programme ou plus simplement pour interagir aisément et rapidement avec des structures de données complexes.
Pour une utilisation en mode script, les instructions à évaluer par l'interprète sont sauvegardées, comme n'importe quel programme informatique, dans un fichier. Dans ce second cas, l'utilisateur doit saisir l'intégralité des instructions qu'il souhaite voir évaluer à l'aide de l'éditeur de texte, prévoir des effets pour afficher les résultats souhaités s'il y en a, puis demander leur exécution à l'interprète.

2. Premiers pas en Python

Un aspect essentiel des langages de programmation est la notion de *type* : tout objet manipulé par Python en possède un, qui caractérise la manière dont il est représenté en mémoire, et dont vont dépendre les fonctions qu'on peut lui appliquer.
En effet, quand nous nous intéresserons plus tard à la représentation en mémoire de certains objets simples, nous constaterons que deux objets de natures distinctes peuvent avoir la même représentation ; dans ce cas, seuls leurs types respectifs vont permettre de les distinguer.

La fonction `type` permet de connaître le type d'un objet. Par exemple, le type *int* est utilisé pour représenter en machine les nombres entiers, le type *str* pour les chaînes de caractères :

```
In [1]: type(5)
Out[1]: int

In [2]: type('LLG')
Out[2]: str
```

L'objet `None` possède lui aussi un type (dont il est le seul représentant) :

```
In [3]: type(None)
Out[3]: NoneType
```

Et même une fonction possède un type : c'est aussi un objet Python.

```
In [4]: type(print)
Out[4]: builtin_function_or_method
```

Par ailleurs, chaque objet possède un *identifiant* (accessible par la fonction `id`) qui est l'adresse mémoire où se trouve stockée la représentation machine de l'objet :

```
In [5]: id(5)
Out[5]: 4297331360

In [6]: id(None)
Out[6]: 4297071472

In [7]: id(print)
Out[7]: 4298509576
```

Dans l'immense majorité des cas nous n'aurons que faire de ces identifiants aussi il n'est pas nécessaire de mémoriser cette fonction. Néanmoins, elle nous sera utile lorsque nous chercherons à comprendre le fonctionnement en mémoire de certains mécanismes simples.

2.1 La représentation des nombres en Python

En mathématique nous connaissons plusieurs sorte de nombres : entiers naturels, entiers relatifs, nombres rationnels, nombres réels, nombres complexes. En Python il existe de base trois types qui peuvent représenter des nombres :

- le type *int* représente des nombres entiers ;
- le type *float* représente des nombres décimaux ;
- le type *complex* représente des nombres complexes.

(Un module additionnel permet de manipuler des nombres rationnels).
Bien entendu, cette représentation est forcément imparfaite : les ensembles de nombres mathématiques sont de cardinal infini or seul un nombre fini d'entre eux peut être représenté en machine car la mémoire est en quantité finie. Pour des raisons analogues les nombres réels dont la représentation décimale est infinie [2] ne peuvent être représentés autrement que par une approximation.

Attention. En mathématique, un nombre entier est aussi un nombre réel et un nombre réel est aussi un nombre complexe. *Ce n'est pas le cas en informatique*, chacun des trois types *int*, *float* et *complex* étant représenté d'une façon spécifique en mémoire. par exemple, le nombre 2 peut être représenté :

- dans le type *int* sous la forme `2`
- dans le type *float* sous la forme `2.0`
- dans le type *complex* sous la forme `2 + 0j`

`x + y`	somme de x et y
`x - y`	différence de x et y
`x * y`	produit de x par y
`x / y`	quotient de x par y
`x ** y`	x puissance y
`x // y`	quotient de la division euclidienne de x par y
`x % y`	reste de la division euclidienne de x par y
`abs(x)`	valeur absolue (ou module) de x

Figure 1 – Les principales opérations sur les nombres.

Il existe des langages de programmation fortement typés : dans un tel langage il est tout bonnement impossible d'effectuer une opération mêlant des objets de types différents (c'est le cas par exemple de Caml, le langage de l'option informatique). Ce n'est heureusement pas le cas du langage Python : si cela s'avère nécessaire pour réaliser une opération, la conversion de type est automatique.

Observons quelques exemples de calculs :

2. Plus précisément ceux dont la représentation *dyadique* (c'est-à-dire en base 2) est infinie.

```
In [1]: 4 * 5
Out[1]: 20

In [2]: 23 / 3
Out[2]: 7.666666666666667
```

```
In [3]: 4 * 5.
Out[3]: 20.0

In [4]: 24 / 4
Out[4]: 6.0
```

Le produit de deux entiers étant toujours entier, le premier calcul ne nécessite donc pas de conversion de type : les deux arguments sont de type *int*, le résultat aussi.

Le quotient de deux entiers n'étant pas toujours entier, le second calcul nécessite de convertir au préalable les deux arguments au type *float*. Le résultat affiché est donc celui de l'opération `23.0 / 3.0`.

Le troisième calcul est le produit d'un objet de type *int* par un objet de type *float*. Le premier objet doit donc être converti au type *float* avant que l'opération ne soit effectuée.
Notez bien que bien que le second opérande ainsi que le résultat représentent des entiers (au sens mathématique), il n'y a pas de conversion automatique au type *int*. En effet, les conversions automatiques ne sont possibles que dans le sens : *int* ⟶ *float* ⟶ *complex*. Dans les autres sens, la conversion doit être explicite (voir plus loin).

Enfin, le dernier calcul montre que même si le résultat de la division est entier (au sens mathématique du terme), le résultat d'une opération faisant intervenir l'opérateur `/` sera toujours de type *float* (ou *complex*). Le résultat affiché est celui de l'opération `24.0 / 4.0`.

Division euclidienne
En informatique on utilise très régulièrement le quotient et le reste d'une division euclidienne : les opérations associées se notent respectivement `//` et `%` :

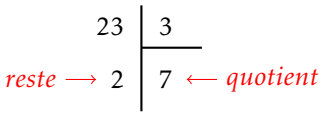

Figure 2 – Quotient et reste de 23 par 3.

```
In [5]: 23 // 3
Out[5]: 7

In [6]: 23 % 3
Out[6]: 2
```

```
In [7]: 24 // 4
Out[7]: 6

In [8]: 24 % 4
Out[8]: 0
```

Ces deux opérations sont à privilégier lorsqu'on souhaite obtenir une réponse sous forme entière puisque le calcul s'effectue exclusivement avec le type *int* [3].

• Conversion explicite de type

Nous avons vu que si cela s'avère nécessaire, l'interprète de commande réalise la conversion implicite de type *int* ⟶ *float* ⟶ *complex*, ce qui est somme toute naturel compte tenu des inclusions mathématiques $\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$. Les conversions réciproques sont *parfois* possibles, mais le résultat n'est jamais garanti [4]. Le nom des fonctions de conversion est simple à retenir : il s'agit tout simplement des noms des types qu'on souhaite obtenir. En voici quelques exemples :

```
In [9]: int(2.3333)
Out[9]: 2

In [10]: int(-2.3333)
Out[10]: -2

In [11]: float(25)
Out[11]: 25.0
```

```
In [12]: complex(2)
Out[12]: (2+0j)

In [13]: float(9999999999999999)
Out[13]: 1e+16

In [14]: int(1e+16)
Out[14]: 10000000000000000
```

Comme on peut le constater, les conversions *float* ⟶ *int* (exemples 9 et 10) mais aussi *int* ⟶ *float* (exemple 13) peuvent conduire à des résultats inexacts sur le plan mathématique.

3. Nous apprendrons plus tard que les opérations sur le type *int* donnent toujours des résultats exacts alors que les calculs sur le type *float* sont toujours entachés d'une marge d'erreur.
4. À utiliser avec circonspection, donc.

Notons enfin qu'il n'est pas possible de convertir un type *complex* en type *float* ou *int* (si nécessaire prendre la partie réelle pour obtenir un objet de type *float*).

```
In [15]: float(2 + 0j)
TypeError: can't convert complex to float

In [16]: (2 + 0j).real
Out[16]: 2.0

In [17]: (2+0j).imag
Out[17]: 0.0
```

• Fonctions mathématiques supplémentaires

Lorsqu'on utilise le langage Python de base, seul un nombre très limité de fonctions est disponible ; heureusement, il existe une multitude de modules additionnels que l'utilisateur peut ajouter en fonction de ses besoins et qui lui fournissent des fonctions supplémentaires. Par exemple, les modules `math` ou `numpy` enrichissent le corpus utilisable des fonctions mathématiques suivantes :

- les fonctions trigonométriques sin, cos, tan ainsi que leurs fonctions réciproques [5] arcsin, arccos, arctan (les angles s'expriment par défaut en radians) ;
- les fonctions exponentielle exp et logarithme log ;
- la fonction racine carrée sqrt.

Sachez qu'il en existe de nombreuses d'autres ; si nécessaire, consulter l'aide en ligne pour les découvrir.

Il existe deux modes d'importation d'un module. Par exemple, pour importer le module numpy on peut :

1. utiliser l'instruction `import numpy`
   Dans ce cas, toutes les fonctions de ce module devront être préfixées du nom du module pour être utilisées : pour calculer un sinus il faudra utiliser la fonction `numpy.sinus`
   Comme les noms des modules peuvent être longs on peut leur donner un nom d'usage plus court en écrivant par exemple :
   ```
   import numpy as np
   ```
   Dans ce cas on calcule un sinus à l'aide de la fonction `np.sin`
2. utiliser l'instruction `from numpy import *`
   Dans ce cas les fonctions n'ont pas besoin d'être préfixées par le nom du module : pour calculer un sinus il suffit d'utiliser la fonction `sin`
   Néanmoins ce mode d'importation n'est pas sans danger si on utilise plusieurs modules différents qui contiennent des fonctions de même nom car dans ce cas, c'est la fonction issue de la *dernière* importation qui sera utilisée [a].
   On peut diminuer ce risque en n'important que les fonctions dont on aura l'usage. Par exemple, si on ne prévoit de n'utiliser que les fonctions trigonométriques on écrira :
   ```
   from numpy import sin, cos, tan
   ```

a. Ce mode d'importation présente un autre défaut : il charge en mémoire l'intégralité des définitions contenues dans ce module, alors que le mode précédent se contente de mémoriser le chemin menant au module en question.

Figure 3 – Les deux modes d'importation d'un module.

S'agissant de fonctions à valeurs numériques, le résultat retourné est le plus souvent un objet de type *float* ou *complex*.

5. Vous en apprendrez les définitions cette année.

```
In [18]: import numpy as np

In [19]: np.sqrt(2)
Out[19]: 1.4142135623730951

In [20]: np.pi
Out[20]: 3.141592653589793

In [20]: np.sin(np.pi)
Out[20]: 1.2246467991473532e-16

In [21]: from math import exp

In [22]: exp(1)                  # la fonction exponentielle du module math
Out[22]: 2.718281828459045

In [23]: np.exp(1)               # la fonction exponentielle du module numpy
Out[23]: 2.7182818284590451
```

2.2 Booléens

Le type *bool* est un type un peu particulier, puisqu'il ne comporte que deux objets : `True` et `False`, qui représentent le vrai et le faux.
À ce type sont associés trois opérateurs : **`not`**, **`and`** et **`or`**, qui sont définis par les tables suivantes :

not	
False	True
True	False

and	False	True
False	False	False
True	False	True

or	False	True
False	False	True
True	True	True

Par ailleurs, un certain nombre d'opérateurs sont définis sur d'autres types (en particulier les types de nombres) et à valeurs dans le type *bool* (voir figure 4).

`x < y`	retourne `True` si $x < y$, `False` sinon
`x > y`	même chose avec la condition $x > y$
`x <= y`	même chose avec la condition $x \leqslant y$
`x >= y`	même chose avec la condition $x \geqslant y$
`x == y`	même chose avec la condition $x = y$
`x != y`	même chose avec la condition $x \neq y$

FIGURE 4 – Opérateurs à valeurs dans le type *bool*.

Ces différents opérateurs peuvent se combiner entre eux et ainsi, on retiendra qu'*évaluer* une expression booléenne n'est pas autre chose que le résultat d'un *calcul*, à l'instar des calculs que l'on effectue sur les nombres.

```
In [1]: (4 + 3) < 11 and not 'alpha' > 'omega'
Out[1]: True

In [2]: (1 + 1 == 3) != ('Henri 4' > 'Louis-le-Grand')
Out[2]: False
```

2.3 Variables

Les données calculées peuvent être mémorisées à l'aide de *variables*. Pour ce faire, il faut attribuer un *nom* à cette variable et lui associer une *valeur* à l'aide de l'opérateur d'affectation =. Le nom de la variable est une suite de lettres (minuscules ou majuscules) et de chiffres, qui doit toujours commencer par une lettre. Il est d'usage de choisir des termes explicites pour faciliter la lecture du code.
Une fois affectée, la valeur de la variable peut être utilisée dans un calcul en faisant référence à son nom. Voici un exemple de calcul de l'aire d'un rectangle dans lequel les noms de variables ont été choisis de manière à rendre le code limpide :

```
In [1]: largeur = 12.45

In [2]: longueur = 42.18

In [3]: aire = longueur * largeur

In [4]: print("l'aire du rectangle est égale à", aire)
l'aire du rectangle est égale à 525.141
```

Notez qu'une affectation réalise un effet (sur la mémoire) et retourne la valeur `None`. La création d'une variable établit un lien entre sa référence (c'est à dire son nom) et un emplacement en mémoire qui contient la valeur de celle-ci.
À l'issue de ce script la situation en mémoire [6] est présentée figure 5.

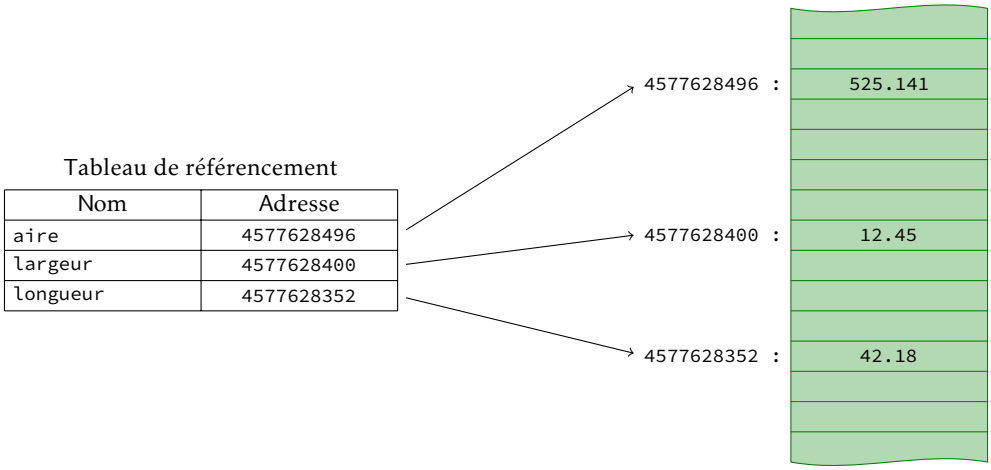

Figure 5 – Variables référencées en mémoire.

En général les utilisateurs que nous sommes n'ont que faire de l'adresse mémoire où sont stockées les valeurs manipulées et celle-ci restera cachée, à moins d'utiliser la fonction `id` qui la renvoie :

```
In [5]: id(largeur)
Out[5]: 4577628400
```

Une fois définie, la valeur d'une variable peut être modifiée, toujours à l'aide de l'opérateur d'affectation. À titre d'exemple, modifions la valeur de la variable `largeur` :

```
In [6]: largeur = 15.7
```

et regardons de nouveau l'identifiant de la variable :

```
In [7]: id(largeur)
Out[7]: 4577628442
```

Il a été modifié, ce qui nous permet de deviner ce qui s'est passé : la nouvelle valeur 15,7 a été stockée à l'adresse 4577628442 et le tableau de référencement modifié en conséquence (illustration figure 6). L'emplacement mémoire contenant l'ancienne valeur 12,45 n'est plus référencé et est réaffecté à l'espace mémoire disponible.

Notons que la valeur de la variable `aire` n'a pas été modifiée. Le calcul effectué ligne 3 utilise les valeurs des variables *au moment où il a été exécuté* et n'établit en aucune manière une liaison particulière entre les variables elles-mêmes.
Ce mécanisme permet de modifier le contenu d'une variable à l'aide de sa propre valeur, comme par exemple :

```
In [8]: longueur = longueur + 1
```

À l'issue de cette instruction la variable `longueur` aura pour valeur 43,18 (et son identifiant sera différent). Modifier une variable à l'aide de sa propre valeur est d'ailleurs une opération tellement fréquente en informatique que cette instruction ligne 6 aurait pu s'écrire plus succinctement : `longueur += 1`.

6. Sur ce schéma, adresses et valeurs sont indiquées sous forme décimale pour des raisons de lisibilité.

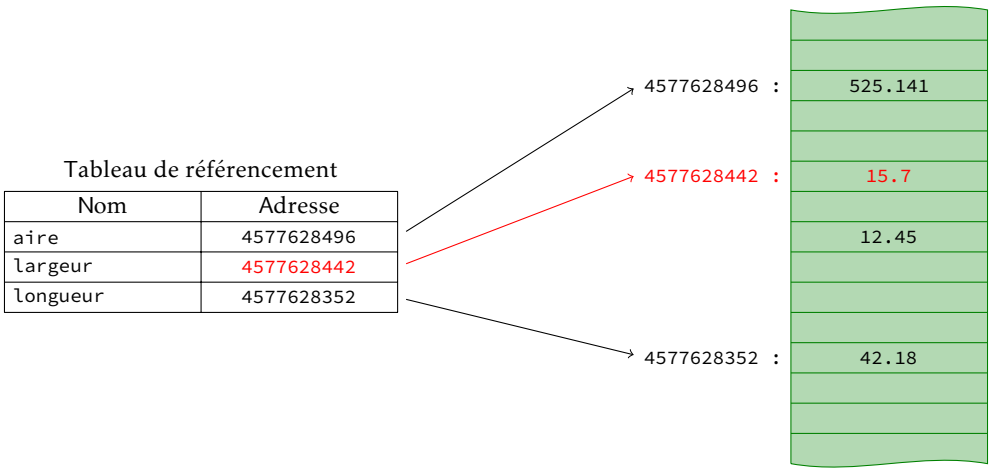

FIGURE 6 – Nouveau référencement d'une variable.

Remarque. Le recyclage de l'espace déréférencé est le fait d'un processus tournant en tâche de fond : le *garbage collector* (ou *ramasse-miettes* en français). Ce programme détermine quels sont les objets en mémoire qui ne peuvent plus être utilisés par l'interprète de commande (parce qu'ils ne sont plus référencés) et ré-affecte l'espace qu'ils occupent à l'espace mémoire disponible. Notons que les langages de plus bas niveau (le C par exemple) ne possèdent en général pas de ramasse-miettes, ce qui oblige le programmeur à allouer et libérer lui-même la mémoire (ce qui en contrepartie permet une gestion plus fine qu'un processus automatique).

Affectations parallèles

Un problème classique auquel est confronté le programmeur débutant est la permutation du contenu de deux variables a et b. En effet, la succession d'opérations :

```
b = a
a = b
```

ne convient pas car ces deux affectations sont effectuées séquentiellement et non pas simultanément. Pour s'en convaincre, il suffit de représenter la modification des référencements après chacune de ces deux affectations (voir figure 7).

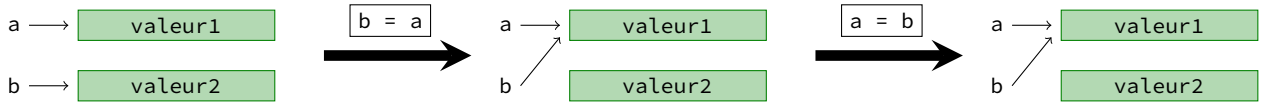

FIGURE 7 – Une permutation incorrecte.

À l'issue de ces deux instructions l'une des deux valeurs n'est plus référencée et l'autre l'est par les deux variables a et b.

Heureusement il est possible en PYTHON d'effectuer plusieurs applications simultanément en séparant les noms et les valeurs correspondantes par des virgules, comme dans l'exemple ci-dessous :

```
In [9]: a, b = 42, 78.5
```

Dès lors, permuter leur contenu devient évident :

```
In [10]: a, b = b, a
```

Exercice 2 S'il n'était pas possible d'effectuer plusieurs affectations simultanément, comment procéderiez-vous pour permuter le contenu de a et b ?

Égalité de valeur, égalité physique
De ce mécanisme résulte l'existence de deux sortes d'égalités entre variables :

- on parle d'*égalité de valeur* lorsque deux variables *a* et *b* référencent la même valeur à deux emplacements mémoires pouvant être différents ;
- on parle d'*égalité physique* lorsque deux variables *a* et *b* référencent le même emplacement mémoire (ce qui implique bien entendu l'égalité de valeur).

Nous connaissons déjà l'opérateur qui teste l'égalité de valeur : il s'agit de l'opérateur == ; l'opérateur qui teste l'égalité physique se nomme `is`.

```
In [11]: a = 257

In [12]: b = 257

In [13]: c = a

In [13]: a == b
Out[13]: True

In [14]: a is b
Out[13]: False

In [14]: a is c
Out[13]: True
```

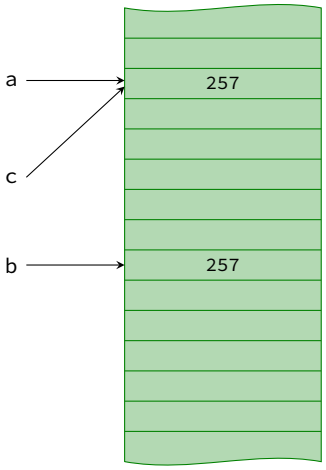

Figure 8 – Différence entre égalité de valeur et égalité physique.

Dans la majorité des cas nous n'aurons besoin que de l'égalité de valeur ; l'égalité physique ne nous sera utile que pour illustrer certaines parties du cours.

2.4 Chaînes de caractères

Outre les types numériques, Python permet la manipulation de données alphanumériques, qu'on appelle des *chaînes de caractères*, et qui correspondent au type *str* (*string* en anglais). On définit une chaîne de caractères en encadrant celle-ci par des ', des ", voire des ''', ces caractères ne faisant pas partie de la chaîne. Le choix est souvent imposé par le contenu : une chaîne contenant un guillemet simple sera encadré par des guillemets doubles, et réciproquement. Par exemple :

```
In [1]: "aujourd'hui"
Out[1]: "aujourd'hui"

In [2]: 'et demain'
Out[2]: 'et demain'
```

Certains caractères spéciaux se définissent à l'aide du caractère d'échappement \ (le *backslash*) ; c'est le cas par exemple du caractère spécial interprété comme un passage à la ligne, représenté par \n :

```
In [3]: print("un passage\n à la ligne")
un passage
 à la ligne
```

(notez que l'espace qui suit le passage à la ligne compte comme un caractère).
Comme on peut le constater, la fonction `print` reconnait de tels caractères et les affiche correctement.

À beaucoup d'égards, une chaîne de caractère ne diffère pas d'une donnée numérique : on peut réaliser des opérations sur les chaînes, même si dans le cas présent ces opérations se réduisent à la concaténation représenté par l'opérateur + et à la duplication par l'opérateur *. Par ailleurs on peut assigner une chaîne de caractères à une variable :

```
In [4]: chn = "Hello "

In [5]: chn += 'world !'     # équivalent à chn = chn + 'world !'

In [6]: print(chn)
Hello world !

In [7]: chn * 3              # équivalent à chn + chn + chn
Out[7]: 'Hello world !Hello world !Hello world !'
```

Attention à bien faire la différence entre la chaîne de caractère `'123'` (de type *str*) et l'entier `123` (de type *int*) ; ces deux objets sont représentés différemment en mémoire et l'opérateur + ne va pas réagir de la même façon :

```
In [7]: '123' + '1'          # concaténation de deux chaînes
Out[7]: '1231'

In [8]: 123 + 1              # addition de deux entiers
Out[8]: 124

In [9]: '123' + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

Le troisième exemple illustre une erreur de typage : il n'existe pas d'opérateur noté + entre une chaîne de caractères et un entier.
Contrairement à ce que nous avons pu observer entre le type *int* et le type *float*, il n'existe pas ici de conversion automatique de type entre le type *int* et le type *str*, mais les fonctions `int` et `str` permettent de réaliser explicitement cette conversion :

```
In [10]: int('123') + 1
Out[10]: 124

In [11]: '123' + str(1)
Out[11]: '1231'
```

Il en est de même de la fonction `float`, qui permet de convertir une donnée adéquate (entier ou chaîne de caractères) en un nombre flottant.

• Accès aux caractères individuels d'une chaîne

Python offre une méthode simple pour accéder aux caractères contenus dans une chaîne : chaque caractère est accessible directement par son rang dans la chaîne en utilisant des crochets. Attention, comme souvent en informatique, le premier caractère de la chaîne est indexé par 0 :

```
In [12]: ch = 'Louis-Le-Grand'

In [13]: ch[4]
Out[13]: 's'

In [14]: ch[0] + ch[6] + ch[9]
Out[14]: 'LLG'
```

Il est aussi possible d'indexer les caractères par des entiers négatifs ce qui revient à compter à partir de la fin (le dernier caractère de la chaîne est alors d'indice −1) :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	o	u	i	s	-	L	e	-	G	r	a	n	d
−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

```
In [15]: print(ch[-8] * 2 + ch[-5])
LLG
```

Notons enfin que la fonction `len` permet de calculer la longueur d'une chaîne de caractères :

```
In [16]: len(ch)
Out[16]: 14
```

Slicing
Plus généralement, la technique sur *slicing* (littéralement le *découpage en tranches*) permet d'extraire une portion d'une chaîne de caractères : il suffit de préciser l'indice de début `i` (qui sera inclus dans le découpage) et l'indice de fin `j` (qui sera exclus) sous la forme `[i:j]`.

```
In [17]: ch[3:-3]
Out[17]: 'is-Le-Gr'
```

Si l'indice `i` n'est pas précisé, il est implicitement pris égal à 0 ; de même si `j` n'est pas précisé il sera égal à la longueur de la chaîne :

```
In [18]: ch[6:] + ch[:6]
Out[18]: 'Le-GrandLouis-'
```

Comme on peut le constater, avec cette technique il est très simple d'extraire les k premiers caractères d'une chaîne : il suffit d'écrire `ch[:k]`. De même, pour extraire les k derniers caractères il suffit d'écrire `ch[-k:]`.

Enfin, le slicing dans sa forme la plus générale prend un troisième paramètre correspondant au *pas* de la sélection : `ch[i:j:k]` retourne tous les caractères dont les indices sont compris entre i (inclus) et j (exclus) et séparés d'un pas k. Par exemple, pour obtenir les caractères d'indices pairs puis les caractères d'indices impairs on écrira :

```
In [19]: ch[::2]
Out[19]: 'LusL-rn'

In [20]: ch[1::2]
Out[20]: 'oi-eGad'
```

Enfin, notons que le pas peut aussi prendre des valeurs négatives, ce qui nous donne un moyen simple d'inverser les caractères d'une chaîne :

```
In [21]: ch[::-1]
Out[21]: 'dnarG-eL-siuoL'
```

(Lorsque le pas est négatif, les valeurs par défaut de `i` et `j` sont respectivement `-1` et `-ln(ch)-1`.)

Exercice 3 Le *mélange de* Monge d'un paquet de cartes numérotées de 1 à $2n$ consiste à démarrer un nouveau paquet avec la carte 1, à placer la carte 2 au dessus de ce nouveau paquet, puis la carte 3 au dessous du nouveau paquet et ainsi de suite en plaçant les cartes paires au dessus du nouveau paquet et les cartes impaires au dessous.
Autrement dit, si le paquet de cartes initial est représenté par la suite $(1, 2, 3, \ldots, 2n)$, son mélange de Monge sera représenté par la suite $(2n, 2n-2, \ldots, 4, 2, 1, 3, 5, \ldots, 2n-3, 2n-1)$.
Réaliser en une ligne Python le calcul d'un mélange de Monge d'une chaîne de caractères `s`.