Chapitre n°1

Structures de données

Table des matières

1 Variable en Informatique **2**
1.1 Structure d'une variable 2
1.2 Création et modification d'une variable 2

2 Types simples **3**
2.1 Types numériques 3
2.1.1 Entiers (`int`) 4
2.1.2 Flottants (`float`) 4
2.2 Booléens (`bool`) 4
2.3 *None* (`NoneType`) 5

3 Types composés **6**
3.1 Classification des variables composées 6
3.2 Conteneur ordonnés (ou séquences) 7
3.2.1 Opérations communes à toutes les séquences 7
3.2.2 Tuples (`tuple`) 8
3.2.3 Listes (`list`) 8
3.2.4 Chaînes de caractères (`str`) 9
3.2.5 *Slicing* de séquences 10
3.3 Dictionnaires (`dict`) 12

4 Conversion de type en Python **13**

5 Une erreur fréquente avec l'instruction `b = a` **14**

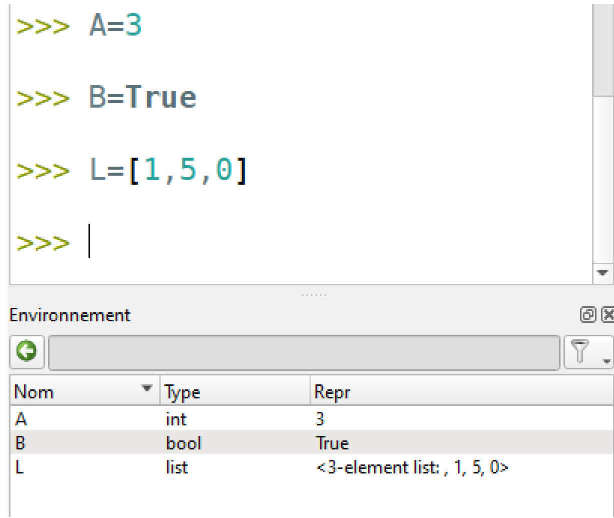
6 Annexe **15**
6.1 Les infinis et formes indéterminées 15
6.2 Les complexes 16
6.3 Sur les chaines de caractères 17

À l'issue de ce cours **17**

1. Variable en Informatique

1.1. Structure d'une variable

Voici trois instructions tapées dans la console Pyzo, ainsi que l'environnement (*Workspace* en anglais) obtenu après cela :



Le Workspace montre qu'il y a trois **variables informatiques** et présente les informations associées à chacune. Pour chaque variable informatique, on associe :

- un **nom** permettant de l'identifier (ici `A`, `B`, `L`) ;
- un **type** de donnée (ici `int`, `bool`, `list`) ;
- une **valeur** (ici `3`, `True`, `[1,5,0]`) ;
- une **adresse mémoire**, repérée par un **identifiant**, et qui n'est pas affichée dans le Workspace.

L'adrese mémoire indique l'endroit où l'ordinateur stocke la valeur de la variable. Lorsqu'on a besoin de la valeur de la variable, Python interroge l'adresse mémoire repérée par l'identifiant de la variable.
Lorsqu'une nouvelle variable est créée, on lui assigne un nouvel espace mémoire (et un nouvel identifiant) dans lequel on stocke sa valeur.
On peut retrouver ces informations en Python à l'aide de fonctions natives `type` et `id` :

Code Python

```
>>> r = 42
>>> r, type(r), id(r)
(42, <class 'int'>, 1351878416)
```

Dans l'exemple ci-dessus, le nom de la variable est `r`, sa valeur est 42, son type est un entier et son identifiant de est 1351878416.

1.2. Création et modification d'une variable

En Python, l'**affectation** d'une variable se fait avec le signe =.
Cela permet de changer la valeur d'une variable et éventuellement son type.

Code Python

```
>>> # Création de variables
>>> a = "la réponse est"
>>> b = 42
>>> a, b
('la réponse est', 42)
>>> # Réaffectation des variables a et b
>>> a = "pouet"
>>> b = a # maintenant b est du type "string"
>>> a, b
('pouet', 'pouet')
```

L'exécution de `var = exp` se fait en deux temps : d'abord, l'expression `exp` est évaluée et donne une valeur. Ensuite, cette valeur est affectée à la variable `var`.

Python permet également d'effectuer des **affectations parallèles** de plusieurs variables. Les affectations parallèles sont simultanées :

Code Python

```
>>> # Affectations parallèles
>>> a, b = "hello", "world"
>>> a
'hello'
>>> b
'world'

>>> # Les affectations parallèles sont simultanées
>>> c, d = 42, c
Traceback [...]
NameError: name 'c' is not defined
```

2. Types simples

Chaque objet [1] utilisé en Python possède un type (accessible à l'aide de la fonction native `type`). Python dispose d'un ensemble de type « simples » [2] définis de manière native : `int`, `float`, `complex`, `bool`, `NoneType`, etc.

2.1. Types numériques

Les **types numériques** de variables informatiques recouvrent tous les types de nombre rencontrés en calcul : on y trouve notamment les **entiers** (`int`), les **flottants** (`float`) et les **complexes** (`complex`). Ces types sont compatibles avec les opérations usuelles (addition, soustraction, multiplication, division, etc.), et leur comportement est particulièrement intuitif.

1. Ce qui inclut les variables, mais également les fonctions, etc.
2. Par opposition aux types composés – voir ci-après.

2.1.1. Entiers (int)

Les entiers (relatifs) sont associés au type **int**. Ils supportent les quatre opérations de base (addition +, soustraction -, multiplication *, division /), l'exponentiation ** et la division euclidienne (quotient // et reste %).

2.1.2. Flottants (float)

Les **flottants** sont une représentation (le plus souvent approchée) des réels utilisée en Informatique.[3] Les flottants supportent les mêmes opérateurs que les entiers. Cela inclut la division euclidienne, ce qui est pratique pour récupérer leur partie entière avec l'opérateur // :

Code Python

```
>>> # Création d'un flottant
>>> a = 13.2
>>> a, type(a)
(13.2, <class 'float'>)

>>> 13.2 // 4.5 # 13.2 = 2 * 4.5 + r avec 0 <= r < 4.5
2.0
>>> 13.2 // 1
13
>>> -13.2 // 1
-14
```

À noter que l'évaluation d'une expression faisant intervenir un mélange d'entiers et de flottants sera un flottant :

Code Python

```
>>> 4 + 5
9
>>> 4.0 + 5
9.0
```

2.2. Booléens (bool)

Un **booléen** est une variable qui ne peut prendre que deux valeurs distinctes : VRAI (True en Python) et FAUX (False en Python). Les variables booléennes permettent de définir des **expressions booléennes**, utilisées notamment lors de la vérification de **conditions**.

Les booléens obéissent aux règles de la **logique booléenne**. Les variables booléennes peuvent être combinées à l'aide d'**opérateurs logiques**, dont les principaux sont :

- l'**équivalence**, qu'on écrit avec un *double égal* == en Python ;
- la **négation** NON, qu'on utilise avec not en Python ;
- la **conjonction** ET, qu'on utilise avec and en Python ;
- la **disjonction** OU, qu'on utilise avec or en Python. Attention c'est un ou *inclusif*.
- la **non équivalence**, qui correspond au "ou exclusif", qu'on utilise avec != en Python.

3. On en reparlera dans un chapitre ultérieur.

Chacun de ces opérateurs agit sur un ou plusieurs booléens pour retourner un autre booléen.

Code Python

```
>>> V = True
>>> F = False

>>> V == F
False
>>> V != F
True
>>> not(V)
False

>>> V and F
False

>>> V or F
True
```

Les opérateurs `==` et `!=` s'appliquent à d'autres types : `0==1` renvoie `False`.

Pour assurer que ces opérations se fassent dans l'ordre voulu, il faut rajouter des parenthèses.

2.3. *None* (`NoneType`)

En Informatique, le mot-clé *None* permet d'identifier une variable dont la valeur n'a pas été affectée (variable « vide ») : l'adresse mémoire est vide[4]. Une variable vide n'accepte aucune opération :

Code Python

```
>>> # Création d'une variable vide
>>> foo = None
>>> foo, type(foo)
(None, <class 'NoneType'>)

>>> # Aucune opération ne marche
>>> foo * 0
Traceback [...]
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
>>> foo + foo
Traceback [...]
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

4. Plus précisément, il y a un identifiant mémoire commun pour toutes les variables vides, mais aucune valeur n'est inscrite dans cette adresse

3. Types composés

3.1. Classification des variables composées

Les **conteneurs** sont des objets pouvant contenir d'autres variables. On y trouve notamment :

- les **listes**, dont le type est `list` ;
- les **chaines de caractères**, dont le type est `string` ;
- les **n-uplets**, dont le type est `tuple` ;
- les **dictionnaires**, dont le type est `dict` ;

En Python, les éléments d'un conteneur sont séparés par des virgules `,` (à l'exception des chaînes de caractères). Chaque type possède ses propres propriétés et opérations, et ce indépendamment des objets qu'ils contiennent. On parle alors de **types composés**.

Ex : on peut concaténer une liste d'entiers avec une liste de flottants, voire même avec une liste de listes. La syntaxe est toujours la même : `L+M`, peu importe ce que contient `L` et `M`.

À noter qu'un conteneur peut contenir des variables de plusieurs types différents, par exemple `L = [3, 1.0, "blabla", [] ]`.

Les conteneurs peuvent être classés selon deux caractéristiques principales :

- le caractère **ordonné** ou non-ordonné du conteneur : un conteneur ordonné possède un ordre intrinsèque. Chaque élément d'un conteneur ordonné peut donc être identifié par son **rang** ou **index** au sein du conteneur ;

⚠ Attention ⚠

En Python, les conteneurs ordonnés sont numérotés à partir du rang 0 (et non pas à partir du rang 1). Si un conteneur `C` possède `n` éléments, `C[0]` est le premier élément et `C[n-1]` sera le dernier (on peut aussi utiliser `C[-1]`).

- le caractère **muable** ou **immuable** du conteneur : il est possible de modifier le contenu d'un conteneur muable après la création de celui-ci, contrairement à un conteneur immuable.

La fonction native `len` permet de déterminer le nombre d'éléments dans un conteneur :

Code Python

```
>>> # Exemple de conteneur (liste)
>>> foo = [7, 2, 42]
>>> len(foo)
3
```

	muable	immuable
ordonné	**listes**	**string**, **tuples**
non ordonné	**dictionnaires**	ensembles (set)

3.2. Conteneur ordonnés (ou séquences)

3.2.1. Opérations communes à toutes les séquences

Chaque élément d'une **séquence** (càd conteneur ordonné) peut être appelé à l'aide de son index placé entre crochets `[ ]` :

Code Python

```
>>> t = (False, None, float("inf"))
>>> t[0]
False
>>> t[1]
None
>>> t[2]
inf
>>> t[3]
Traceback [...]
IndexError: tuple index out of range
```

À noter qu'un élément d'une séquence peut être identifié par un index négatif (l'index -1 représentant le dernier élément de la séquence) :

Code Python

```
>>> t = (False, None, float("inf"))
>>> t[-1]
inf
>>> t[-2]
None
>>> t[-3]
False
>>> t[-4]
Traceback [...]
IndexError: tuple index out of range
```

Toutes les séquences supportent l'**opérateur de concaténation** + :

Code Python

```
>>> L1 = [12, 15, 29]
>>> L2 = [42, 13, 50]
>>> L1 + L2
[12, 15, 29, 42, 13, 50]
>>> L2 + L1
[42, 13, 50, 12, 15, 29]
>>> L1 + L1
[12, 15, 29, 12, 15, 29]
```

Nota : on peut uniqement concaténer des séquences de même type.

3.2.2. Tuples (**tuple**)

Les **tuples** (ou n-uplets) sont des séquences immuables déclarées à l'aide de parenthèses ( ) :

Code Python

```
>>> # Création d'un tuple
>>> T = (12, 15, 29)
>>> T, type(T)
((12, 15, 29), <class 'tuple'>)

>>> # Les tuples sont des séquences immuables
>>> T[0] = 42
Traceback [...]
TypeError: 'tuple' object does not support item assignment
```

3.2.3. Listes (**list**)

Les **listes** sont des séquences muables déclarées à l'aide de crochets [ ] :

Code Python

```
>>> L = [12.3, 15.2, 29.7]
>>> L, type(L)
([12.3, 15.2, 29.7], <class 'list'>)

>>> # Les listes sont des séquences muables
>>> L[0] = 3.14
>>> L
[3.14, 15.2, 29.7]
```

Une liste pour être étendue à l'aide de la méthode **append**. L' (unique) élément passé en argument est ajouté à la fin de la liste considérée :

Code Python

```
>>> L = [12, 15, 29]

>>> # Extension de la liste à l'aide de la méthode append
>>> L.append(42)
>>> L
[12, 15, 29, 42]
```

Rappel : on peut aussi *concaténer* deux listes avec le symbole +.

Code Python

```
>>> L = [12,15,29]
>>> L = L + [53,72]
>>> L
[12, 15, 29, 42, 53, 72]
```

En Python, on peut générer des listes de manière efficace et compacte par **compréhension** à partir d'un conteneur (et en particulier d'un objet de type `range`). La syntaxe est la suivante :

[⟨ `f(var)` ⟩ `for` ⟨`var`⟩ `in` ⟨Conteneur⟩]

où `f(var)` est une expression qui peut dépendre de `var`. On peut faire un parallèle avec les suites :

[$\langle u_k \rangle$ `for` $\langle k \rangle$ `in` ⟨`range(n)`⟩]

permet de construire la liste $[u_0, u_1, \dots, u_{n-1}]$. Voici quelques exemples de construction de listes en compréhension :

Code Python

```
>>> # Création d'un conteneur
>>> conteneur = [3, 0, -5, 1]

>>> # Création de listes en compréhension
>>> L = [k + 1 for k in conteneur]
>>> L
[4, 1, -4, 2]

>>> M = [k ** 2 for k in conteneur]
>>> M
[9, 0, 25, 1]
```

À noter que le conteneur peut être une liste, un *string*, un tuple, ou encore un objet de type range.

Code Python

```
>>> [lettre for lettre in "blabla"]
['b', 'l', 'a', 'b', 'l', 'a']
>>> [5-k for k in range(5)] # liste des u_k = 5-k pour k allant de 0 à 4
[ 5, 4, 3, 2, 1 ]
```

3.2.4. Chaînes de caractères (`str`)

Les **chaînes de caractères** (ou *string*) permettent de représenter du texte. Ce sont des séquences immuables déclarées à l'aide de *quotes* `' '` (apostrophes) ou de double-*quotes* `" "` (guillemets). Contrairement aux autres conteneurs, les éléments d'une chaîne de caractères ne sont pas séparés par des virgules lors de sa création :

Code Python

```
>>> S = "Devine si tu aimes"
>>> S, type(bar)
('Devine si tu aimes', <class 'str'>)
>>> len(S) # nombre de lettres dans S
17
```

```
>>> # Les chaînes de caractères sont des séquences
>>> S[0]
'D' # 'D' et 'd' ne sont pas pareils !
>>> S[-1]
's'

>>> # Les chaînes de caractères sont immuables
>>> S[2] = "p"
Traceback [...]
TypeError: 'str' object does not support item assignment
```

3.2.5. *Slicing* de séquences

Le ***slicing***, dans Python, permet de générer une nouvelle séquence par extraction d'une autre : on sélectionne les indices qu'on veut conserver selon le schéma suivant.

`L[ debut : fin ]`

Ceci est la liste des éléments de `L` qui part de l'indice `debut` inclus jusqu'à l'indice `fin` exclu.

Par exemple, si $\texttt{L} = [u_0, u_1, \dots, u_n]$ alors

$$\texttt{L}[d : f] \quad \text{retourne} \quad [u_d, u_{d+1}, \dots, u_{f-1}] \quad \text{càd} \quad [u_k \quad \text{for} \quad k \in \mathbb{N} \text{ "tel que" } d \leq k < f]$$

- Si $f \leq d$, alors $\texttt{L}[d : f]$ est vide.
- $\texttt{L}[\,: f]$ retourne $[u_0, u_1, \dots, u_{f-1}]$. Si on omet l'argument d, il sera pris comme égal à 0. On commence alors au tout début de la liste.
- $\texttt{L}[d :\,]$ retourne $[u_d, u_{d+1}, \dots, u_n]$. Si on omet l'argument f, il sera pris comme égal à `len(L)`, ici $n + 1$. Cela permet d'aller jusqu'à la fin et d'inclure le dernier élément.
- `L[ : ]` est la combinaison des deux ci-dessus. On prend alors la totalité de la liste. L'instruction `M = L[ : ]` est similaire à `M = L.copy()`.

Code Python

```
>>> List = ["a", "b", "c", "d", "e"]

>>> List[0 : 3]
['a', 'b', 'c']
>>> List[2 : ]
['c', 'd', 'e']
>>> List[ : 2]
['a', 'b']
>>> List[1 : 2]
['b']
>>> List[ : ]
['a', 'b', 'c', 'd', 'e']
>>> List[3 : 3]
[ ]

>>> # debut et fin peuvent être négatifs : L[-1] est le dernier élément,
```

```
L[-2] l'avant-dernier etc.
>>> List[-2 : -1]
['d']
>>> List[-2 : ] # tout à partir des deux derniers
['d', 'e']
>>> List[ : -2] # tout sauf les deux derniers
['a', 'b', 'c']
>>> List[1 : -1] # tout à partir du 1er et sauf le dernier
['b', 'c', 'd']
```

Dans un *slicing*, on peut rajouter un troisième argument `pas` $\in \mathbb{Z}^*$:

`L[ debut : fin : pas ]`

Ceci est la liste des éléments de `L` qui part de l'indice `debut` inclus, qui "saute" de `pas` en `pas`, jusqu'à ce qu'on arrive ou dépasse l'indice `fin`, qui est exclu.

Par exemple, si $\texttt{L} = [u_0, u_1, \dots, u_n]$ alors

$\texttt{L}[d : f : p]$ retourne $[u_d, u_{d+p}, \dots, u_{d+mp}]$ avec $m \in \mathbb{N}$ tel que $d + mp < f \leq d + (m+1)p$
càd $[u_k$ for $k \in \mathbb{N}$ "tel que" $d \leq k < f$ et $k = d + ip$ avec $i \in \mathbb{N}]$

- $\texttt{L}[d : f : 1]$ est la même chose que $\texttt{L}[d : f]$.
- $\texttt{L}[d : f : 2]$ prend un élément sur 2 à partir de d (inclus) jusqu'à l'élément d'indice $f - 1$ ou $f - 2$ (on s'arrête "juste avant" f).
- On peut prendre p négatif. Dans ce cas on "recule" au lieu d'avancer (en sautant de $|p|$ en $|p|$). L'élément d'indice f est toujours exclu.
- On peut omettre d et/ou f.
  - Si $p > 0$, alors $\texttt{L}[d : : p]$; $\texttt{L}[: f : p]$ et $\texttt{L}[: : p]$ retournent le même résultat que dans le cas sans p qu'on a vu ci-dessus, mais on saute de p en p.
  - Si $p < 0$, alors si d est omis on commence par le dernier élément (inclus) et si f est omis on va jusqu'au début de la liste (premier élément *inclus*).

CODE PYTHON

```
>>> # Exemple de séquence (tuple)
>>> Tup = ("a", "b", "c", "d", "e")

>>> Tup[ : 2 : 1]
('a', 'b')
>>> Tup[2 : : 1]
('c', 'd', 'e')
>>> Tup[0 : : 2]
('a', 'c', 'e')
>>> Tup[1 : : 2]
('b', 'd')
>>> Tup[0 : : 17]
('a')
>>> Tup[3 : : -1] # on part de l'indice 3 et on recule de 1 en 1
```

```
('d', 'c', 'b', 'a')
>>> Tup[3 : 0 : -1] # l'élément d'indice 0 est exclu
('d', 'c', 'b')
>>> Tup[4 : 1 : -2] # on prend les indices 4, 2 et on s'arrête car 0≤1
('e', 'c')
>>> Tup[-3 : : -1]
('c', 'b', 'a')
>>> Tup[ : : -1]
('e', 'd', 'c', 'b', 'a')
```

A noter : un *slicing* peut s'effectuer sur toute séquence (i.e. conteneur ordonné). En particulier, cela marche pour des tuples (comme ci-dessus) et des *string* (cf ci-dessous). Attention, on utilise toujours les crochets `[ ]` pour du *slicing*.

Code Python

```
>>> # Exemple de séquence (string)
>>> Str = "Saper"

>>> Str[ : 2]
'Sa'
>>> Str[2 : ]
'per'
>>> Str[0 : : 2]
'Spr'
>>> Str[ : : -1]
'repaS'
```

3.3. Dictionnaires (`dict`)

Les **dictionnaires** sont des conteneurs non-ordonnés muables qu'on déclare à l'aide d'accolades `{ }`. Chaque élément d'un dictionnaire est construit selon le modèle `clef:valeur`. L'identifiant **clef** (unique pour un dictionnaire donné, et nécessairement immuable) et d'une **valeur** associée à cette clef (qui est muable).

Code Python

```
>>> # Création d'un dictionnaire
>>> foo = {"a": 42, 7.6: [3, 5]}
>>> foo, type(foo)
({'a': 42, 7.6: [3, 5]}, <class 'dict'>)

>>> # Appel d'un élément d'un dictionnaire à partir de sa clef
>>> foo["a"]
42
>>> foo[7.6]
[3, 5]

>>> foo[0] # 0 n'est pas une clé, erreur
```

```
Traceback [...]
KeyError: 0
>>> foo[42]
Traceback [...]
KeyError: 42 # 42 n'est pas une clé, erreur

>>> # Ajout / modification d'un élément dans un dictionnaire
>>> foo["a"] = 63
>>> foo
{'a': 63, 7.6: [3, 5]}
>>> foo[0] = "pouet"
>>> foo
{'a': 63, 7.6: [3, 5], 0: 'pouet'}
>>> # Suppression d'un élément présent dans un dictionnaire
>>> del(foo["a"])
>>> foo
{7.6: [3, 5], 0: 'pouet'}
>>> del(foo["bla"])
Traceback [...]
KeyError: 'bla'
```

4. Conversion de type en Python

Python est un langage qui peut faire des conversions automatiques pour que des opérations aient un sens : si on réalise des opérations qui impliquent des entiers et des flottants, les entiers seront automatiquement convertis en flottants et traités comme tels dans le calcul (le résultat sera donc exprimé sous la forme d'un flottant). On a vu plus haut l'exemple suivant :

Code Python

```
>>> 4 + 2
6
>>> 4 + 2.0
6.0
```

Il est parfois nécessaire de modifier explicitement (et durablement) le type d'une variable : on parle alors de **conversion de type**. En Python, si on veut convertir une expression vers le type `int`, on utilise la fonction `int(...)`, idem pour les autres types.

Code Python

```
>>> # Conversion entier -> flottant
>>> a = float(42)
>>> a, type(a)
(42.0, <class 'float'>)

>>> # Conversion flottant -> chaîne de caractères
>>> b = str(42.0)
>>> b, type(b)
```

```
('42.0', <class 'str'>)

>>> # Conversion chaîne -> liste
>>> L = list("42.0")
>>> L, type(L)
(['4', '2', '.', '0'], <class 'list'>)

>>> # Conversion impossible : liste -/-> entier
>>> c = int(L)
Traceback [...]
TypeError: int() argument must be a string, a bytes-like object or a
number, not 'list'
```

Comme le montre l'exemple ci-dessus, certaines conversions sont impossibles : Python ne connait pas de façon "naturelle" de convertir une liste en entier.

5. Une erreur fréquente avec l'instruction `b = a`

Si `L` et `M` sont des listes,

Il y a une différence entre `M = L` et `M = L+[]` ! Elle est importante en pratique !

Si `var1` est une variable de type quelconque, l'instruction `var2 = var1` (prise littéralement, il ne doit y avoir rien d'autre qu'une variable `var1` à droite du =) créé (ou modifie) `var2` MAIS lui donne le même identifiant que `var1`. Cela veut dire que `var2` et `var1` pointent vers la même adresse mémoire.

Code Python

```
>>> a=0
>>> b=a
>>> a,id(a)
(0, 140710457255728)
>>> b,id(b)
(0, 140710457255728)
```

En pratique, ce n'est pas gênant pour les types simples, les chaines de caractères (car elles sont immuables) ou encore les tuples (idem).

Code Python

```
>>> # Création de la variable <foo>
>>> foo = "hello"

>>> # Copie du contenu de la variable <foo> dans la variable <bar>
>>> bar = foo


>>> # Modification du contenu de la variable <foo>
>>> foo = "world"
```

```
>>> # La variable <bar> n'a pas été modifiée lors de la modification
de la variable <foo>
>>> foo, bar
('world', 'hello')
```

Mais le fait que `b = a` pointe vers la même adresse mémoire pose problème pour les types composés muables, à savoir les listes et les dictionnaires.

Code Python

```
>>> # Création de la liste <foo>
>>> foo = [1, 2, 3]

>>> # Copie de la liste <foo> dans la variable <bar>
>>> bar = foo

>>> # Modification du contenu de la liste <foo>
>>> foo[0] = 42

>>> # La liste <bar> n'est pas indépendante de la liste <foo>
>>> foo, bar
([42, 2, 3], [42, 2, 3])
```

⚠ Attention ⚠

Si `L` et `M` sont des listes, la syntaxe `M = L` ne créé pas une nouvelle liste, toute modification sur `L` est répercutée sur `M`.
Pour créer une copie de `L` utiliser `M = L.copy()`, ou `M = L + []` à la place.

6. Annexe

6.1. Les infinis et formes indéterminées

En Python, les flottants permettent également de représenter les deux infinis ($+\infty$ et $-\infty$) à l'aide de la constante `inf`, ainsi que les formes indéterminées (*NaN* pour « *Not a Number* ») à l'aide de la constante `nan`. Ces objets se comportent de la manière attendue :

Code Python

```
>>> # Manipulation des infinis
>>> foo = float("inf") # float convertit la chaine "inf" en + infini.
>>> foo, type(foo)
(inf, <class 'float'>)

>>> foo + 42
inf

>>> -foo
-inf
```

```
>>> # Manipulation des formes indéterminées
>>> bar = float("nan") # idem, c'est une conversion
>>> bar, type(bar)
(nan, <class 'float'>)

>>> foo - foo
nan
```

6.2. Les complexes

Python permet également de traiter les nombres complexes de manière native. La syntaxe générale d'un nombre complexe en Python repose sur l'utilisation du j complexe (notation issue de la Physique) :

Code Python

```
>>> # Création d'un nombre complexe
>>> foo = (13+2j)
>>> foo, type(foo)
((13+2j), <class 'complex'>)

>>> # Quelques opérateurs usuels...
>>> (13+2j) + (4+5j)
(17+7j)
>>> (13+2j) * (4+5j)
(42+73j)
>>> 1j ** 3
(-0-1j)
```

Il est possible de récupérer les flottants représentant respectivement les parties réelle et imaginaire d'un nombre complexe à l'aide des méthodes `real` et `imag` :

Code Python

```
>>> foo = (13+2j)
>>> foo.real
13.0
>>> foo.imag
2.0
```

À noter que l'évaluation d'une expression faisant intervenir un mélange d'entiers, de flottants et de nombre complexes sera un nombre complexe :

Code Python

```
>>> 4 + 5.0 + (6+0j)
(15+0j)
```

6.3. Sur les chaines de caractères

Les chaînes de caractères possèdent de nombreuses méthodes permettant de faciliter la manipulation de texte, dont voici quelques exemples :

Code Python

```
>>> S = "voiCi un eXemPle de texTe"
>>> S.lower()
'voici un exemple de texte'
>>> S.upper()
'VOICI UN EXEMPLE DE TEXTE'
>>> S.swapcase()
'VOIcI UN ExEMpLE DE TEXtE'
```

À noter également qu'il existe un certain nombre de **caractères d'échappement** utiles pour la mise en forme de texte. Par exemple :

- la chaîne de caractères `"\n"` code pour un retour à la ligne ;
- la chaîne de caractères `"\t"` code pour une tabulation ;
- etc.

À l'issue de ce cours

À l'issue de ce cours, je suis capable :

- ☐ de créer et de (ré)affecter une variable ;
- ☐ de coder des expressions à partir de variables numériques (entiers, flottants, nombres complexes) et d'opérateurs usuels ;
- ☐ de coder des expression booléennes à partir de variables booléennes et d'opérateurs logiques usuels ;
- ☐ d'identifier la nature (ordonné ou non, muable ou non) d'un conteneur ;
- ☐ d'appeler un élément d'un conteneur ;
- ☐ de déterminer le nombre d'éléments dans un conteneur ;
- ☐ de définir et d'utiliser les opérateurs usuels associés aux séquences (listes, tuples, chaînes de caractères) ;
- ☐ de générer une séquence par *slicing* ;
- ☐ de générer une liste en compréhension ;
- ☐ de réaliser une copie indépendante d'une liste constituée de variables de type(s) simple(s) (par compréhension ou par *slicing*) ;
- ☐ de manipuler un dictionnaire ;
- ☐ de réaliser une conversion de type.