TYPES ET VARIABLES

B. Landelle

Table des matières

I Types simples

1 Les entiers

Définition 1. *Les entiers relatifs sont codés en python par le type* `int`.

Expérimentation :

```
>>> 1
1
>>> type(1)
<class 'int'>
>>> type(-1)
<class 'int'>
>>> type(12345678901234567890)
<class 'int'>
```

Remarque : En apparence, le traitement de petits entiers (petits en valeur absolue) est identique à celui de grands entiers. La réalité est plus complexe. Les grands entiers sont découpés en tableau d'entiers courts ce qui rend tous les calculs sur ces nombres beaucoup plus lents. Ce processus est transparent pour l'utilisateur, sauf si le temps de traitement devient un élément critique pour l'usage en cours.

Théorème 1 (Division euclidienne). *Soit* $(a, b) \in \mathbb{Z} \times \mathbb{Z}^*$.

$$\text{Il existe un unique couple } (q, r) \in \mathbb{Z} \times [\![0 \,;\, |b| - 1]\!] \text{ tel que } a = b \times q + r.$$

Le terme q *est appelé* quotient *et le terme* r *est appelé* reste.

Proposition 1. *Le langage python permet d'effectuer les opérations arithmétiques habituelles sur les entiers* `int` *:*

- *addition* `+`*, soustraction* `-`*,*
- *multiplication* `*`*,*
- *puissance ou exponentiation* `**`*,*
- *valeur absolue* `abs`*,*
- *quotient de division euclidienne* `//`*, reste de division euclidienne* `%`*,*
- *etc.*

Expérimentation :

```
>>> 2-3
-1
>>> 2**3
8
>>> 7//2 # quotient de 7 divisé par 2
3
>>> 7%2 # reste de 7 divisé par 2
1
```

Exercice : Quelle instruction utiliser pour savoir si 3803 divise 123456789 ?

Corrigé : On regarde si le reste de la division euclidienne est nul :

```
>>> 123456789%3803
0
```

Les opérations mathématiques rigoureusement parenthensées sont sans ambiguïtés avec un ordre de priorité entre les opérations complètement déterminé. Sans parenthèse, python ne relève pas d'erreur de syntaxe mais applique ses règles de priorité qu'on appelle *règles de précédence*.

Expérimentation :

```
>>> (2**3)*(3**2)
72
>>> 2**(3*(3**2))
134217728
>>> 2**(3*3**2)
134217728
>>> 2**3*3**2
72
```

Proposition 2 (Règles de précédence des opérateurs sur les entiers). *Dans une expression reliant des entiers et des opérations, Python calcule par priorité décroissante :*

- *les puissances,*
- *les multiplications, les quotients et restes de division euclidienne,*
- *les additions et soustractions.*

Remarque : En cas de doute, on recommande l'usage des parenthèses ...

2 Les flottants

Définition 2. *Un nombre en format* `float` *suivant la norme IEEE 754 est codé par*

$$(-1)^s \times \mathrm{M} \times 2^{\mathrm{E}-1023}$$

avec $s \in \{0,1\}$, $\mathrm{M} = 1 + \sum_{i=1}^{52} \frac{m_i}{2^i}$ *où les* $m_i \in \{0,1\}$ *et* $\mathrm{E} \in [\![0\,;\, 2046]\!]$. *Le nombre décimal* M *est appelé* mantisse.

Remarque : Le format `float` est le format de nombres qu'on utilisera dans le cadre d'expérimentations avec relevé de mesures numériques.

Expérimentation :
La saisie d'un nombre décimal est automatiquement traitée en format `float`. Ce format possède des limitations intrinsèques et la plupart des nombres ne sont pas codés fidèlement en flottant.

```
>>> type(0.1)
<class 'float'>
>>> 0.1+0.2
0.30000000000000004
```

L'utilisateur peut identifier un flottant à la présence du symbole . (en lieu et place de la virgule comme on a l'usage habituellement). Le zéro en tant qu'unité seule ou en tant que dernière décimale peut être omis.

```
>>> .1
0.1
```

Un nombre, même entier, saisi avec virgule entraîne un traitement au format `float`.

```
>>> 1.
1.0
>>> type(1.)
<class 'float'>
```

Le module `numpy` est incontournable en calcul numérique. Il est d'usage de l'importer de la manière suivante :

```
>>> import numpy as np
```

L'intitulé `np` désigne un alias de `numpy`, un nom abrégé pour faire référence au module `numpy`. Les constantes mathématiques usuelles comme π ou e sont évidemment au format `float`. Elles ne sont pas présentes nativement dans le noyau mais figurent dans le module `numpy`. Les puissances de 10 sont également au format `float` (même des puissances positives, ce qui n'est pas évident).

```
>>> np.pi
3.141592653589793
>>> type(np.pi)
<class 'float'>
>>> np.e
2.718281828459045
>>> type(np.e)
<class 'float'>
>>> 1e1
10.0
>>> type(1e1)
<class 'float'>
>>> 4e-3
0.004
```

Proposition 3. *Le langage python permet d'effectuer sur les flottants les opérations arithmétiques habituelles :*

- *addition* `+`*, soustraction* `-`*,*
- *multiplication* `*`*, division* `/`*,*
- *puissance ou exponentiation* `**`*,*
- *valeur absolue* `abs`*,*
- *racine carrée (*square root*)* `np.sqrt`*,*
- *partie entière* `np.floor`*,*
- *fonctions trigonométriques* `np.cos, np.sin, np.tan`*,*
- *logarithme* `np.log`*, exponentielle* `np.exp`*,*
- *etc.*

Rappel : La partie entière notée $\lfloor \cdot \rfloor$ est définie par

$$\forall x \in \mathbb{R} \qquad \lfloor x \rfloor = \text{Max}\,\{n \in \mathbb{Z}, n \leqslant x\}$$

Pour x réel, la partie entière $\lfloor x \rfloor$ est l'unique entier relatif vérifiant

$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$$

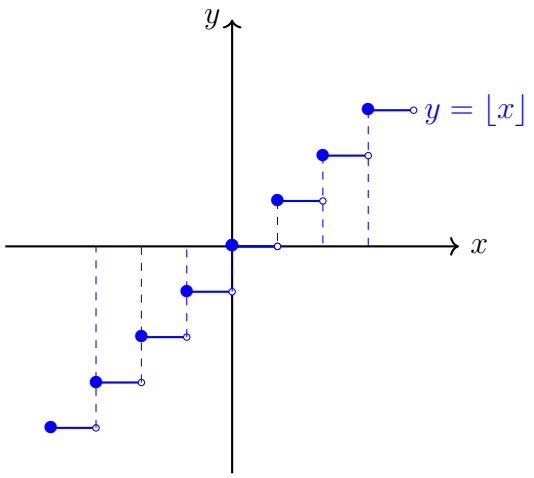

FIGURE 1 – Graphe de la partie entière

Expérimentation :

```
>>> 1/2
0.5
>>> 6/3
2.0
>>> np.sqrt(2)
1.4142135623730951
>>> np.sqrt(2)**2
2.0000000000000004
>>> np.sin(np.pi)
1.2246467991473532e-16
>>> np.cos(np.pi)
-1.0
>>> np.cos(1e10*np.pi)
0.99999999999749267
>>> np.cos(1e100*np.pi)
-0.98980167370179184
```

On observe que l'instruction division renvoie un résultat flottant même si les valeurs saisies et le résultat sont entiers. On observe également les imprécisions numériques intrinsèques au format flottant. Le calcul flottant est un calcul numérique et non un calcul formel. Le dernier résultat numérique est aberrant.

Comme avec les entiers, python possède des règles de priorité qui s'appliquent sur les opérations réalisées sur des flottants écrites sans parenthesage.

Proposition 4 (Règles de précédence des opérateurs sur les flottants). *Dans une expression reliant des flottants et des opérations, python calcule par priorité décroissante :*

- *les puissances,*
- *les multiplications, les divisions,*
- *les additions et soustractions.*

Expérimentation :

```
>>> 2.**3*3.**2
72.0
```

Remarque : Comme pour les entiers, on recommande l'usage des parenthèses pour s'épargner d'apprendre les règles de précédence.

Une propriété notable du format flottant est son caractère absorbant.

Définition 3 (Absorbance du format `float`). *Si une expression numérique contient un nombre au format* `float` *ou si un résultat intermédiaire ou final est de type* `float`*, celui-ci l'emporte sur les autres formats et détermine le type final du résultat. Le type* `float` *présente un caractère* absorbant.

Expérimentation :

```
>>> 0+1.
1.0
>>> 1+0.
1.0
>>> 1/1
1.0
```

Le basculement vers le format float peut s'avérer bloquant, notamment si l'on sort des limites de codage.

```
>>> 2**1024
1797693134.....37216
>>> 2.**1024
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    2.**1024
OverflowError: (34, 'Result too large')
```

L'instruction `float` permet la conversion (sous réserve de compatibilité) vers le format flottant.

```
>>> float(1)
1.0
>>> float(2**1024)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    float(2**1024)
OverflowError: int too large to convert to float
>>> float(2**1023)*2
inf
```

On peut convertir des flottants en entiers avec l'instruction `int` mais le résultat ne correspond pas exactement à la partie entière.

```
 >>> int(3.6)
3
>>> np.floor(3.6)
3.0
>>> int(-1.5)
-1
>>> np.floor(-1.5)
-2.0
```

Exercice : Prédire le résultat des instructions suivantes :

```
>>> type(2/1)
>>> int(3/2)
>>> type(2**1024/2)
>>> type(2.**1024/2)
```

Corrigé :

```
>>> type(2/1)
<class 'float'>
>>> int(3/2)
1
>>> type(2**1024/2)
<class 'float'>
>>> type(2.**1024/2)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    type(2.**1024/2)
OverflowError: (34, 'Result too large')
```

3 Les booléens

Définition 4. *Les* booléens *sont des types de constantes sous python pouvant prendre deux états :* `True` *et* `False`.

Remarque : Les booléens ne sont pas une spécificité de python. On les retrouve dans tous les autres langages de programmation.

Expérimentation :

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Proposition 5. *En python, les relations binaires* $=, \neq, <, >, \leqslant, \geqslant$ *codées respectivement par les instructions* `==`, `!=`, `<`, `>` , `<=`, `>=` *renvoient des booléens.*

Expérimentation :

```
>>> 1>0
True
>>> 1<2
True
>>> 1>2
False
>>> 1==2
False
>>> 1!=2
True
```

Quelques tests moins prévisibles :

```
>>> 1>0.
True
>>> 1==1.
True
>>> 1==True
True
>>> 0==False
True
```

Python teste l'égalité quantitative et non qualitative, le type n'influant pas dans le résultat. Le booléen `True` est traité numériquement comme 1 et `False` comme 0.

```
>>> 1+False
1
>>> True+0.
1.0
>>> float(False)
0.0
>>> int(True)
1
```

Réciproquement, on peut convertir des entiers ou flottants en booléens avec l'instruction `bool` selon la règle suivante : zéro devient `False` et un objet qui n'est pas zéro devient `True`.

```
>>> bool(0.)
False
>>> bool(1)
True
>>> bool(np.sqrt(2))
True
```

Définition 5. *Les opérations sur les booléens sont les opérations logiques usuelles :*
- *la négation réalisée par l'instruction* `not` *;*
- *la conjonction (le* `et` *logique) réalisée par l'instruction* `and` *;*
- *la disjonction (le* `ou` *logique) réalisée par l'instruction* `or`*.*

p	q	not p	p and q	p or q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

SCHÉMA - Tables de vérité

Expérimentation :

```
>>> 2>1 and 3>2
True
>>> 2>1 and 3<2
False
>>> 2>1 or 3<2
True
>>> 2>1 and not 3<2
True
```

Quelques tests moins prévisibles :

```
>>> 1==2 and 1==1 or 1!=2
True
>>> 1==2 and (1==1 or 1!=2)
False
>>> not False and False
False
>>> not (False and False)
True
>>> not True and True or True
True
>>> not True and (True or True)
False
```

Proposition 6 (Règles de précédence des opérateurs sur les booléens). *Dans une expression reliant des booléens et des opérations, python calcule par priorité décroissante :*

- *les négations*
- *les conjonctions,*
- *les disjonctions.*

Remarque : Python ne calcule pas l'intégralité d'une expression quand la valeur de celle-ci est complètement déterminée par le début de cette expression (la suite de l'expression peut donc éventuellement être incalculable). Ce comportement s'appelle l'évaluation *paresseuse* (*lazy evaluation* en anglais).

```
>>> True or 1/0==1
True
>>> False and 1/0==1
False
```

Exercice : Prédire le résultat des instructions suivantes :

```
>>> 1>2 and 2>3 or 2>1
>>> not 1>2 and 2>3
>>> not 2>1 or 2>3 and 2>1
>>> 2<1 and not 2>1 or 2>1
```

Corrigé :

```
>>> 1>2 and 2>3 or 2>1
True
>>> not 1>2 and 2>3
False
>>> not 2>1 or 2>3 and 2>1
False
>>> 2<1 and not 2>1 or 2>1
True
```

Exercice : Les instructions `max` et `min` renvoient respectivement le maximum et le minimum de deux nombres. Écrire avec des opérateurs sur les booléens les tests suivants :

```
1>max(2,3)
1<max(2,3)
1<max(2,min(3,4))
```

Corrigé : On saisit :

```
>>> 1>2 and 1>3
False
>>> 1<2 or 1<3
True
>>> 1<2 or 1<3 and 1<4
True
```

II Variables

1 Définition

Définition 6. *Une* variable *désigne une étiquette vers un emplacement mémoire servant au stockage d'un certain type d'objet.*

Proposition 7. *L'instruction* `id(var)` *d'argument une variable* `var` *renvoie l'adresse en mémoire de l'objet étiqueté par la variable* `var`.

Expérimentation :

```
>>> a=1
>>> a
1
>>> id(a)
1695921840
```

FIGURE 2 – Variable `a` étiquetée sur l'entier 1

L'instruction `a=1` crée une variable `a` qui est étiquetée sur l'entier 1. On dit fréquemment qu'on *affecte* ou qu'on *assigne* à la variable `a` la valeur `1`. Cette notion d'*affectation* est très pertinente dans certains langages, un peu moins en python où la notion d'étiquetage est plus fidèle à la réalité.

```
>>> a=1
>>> b=a
>>> b
1
>>> id(a)
1695921840
>>> id(b)
1695921840
```

FIGURE 3 – Variables a et b étiquetées sur le même objet

L'instruction `b=a` créée une nouvelle variable étiquetée sur le même objet que celui étiqueté par la variable `a`.

```
>>> b=2
>>> b
2
>>> id(a)
1695921840
>>> id(b)
1695921872
```

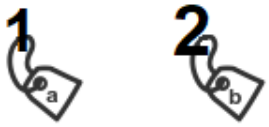

FIGURE 4 – Variables `a` et `b` étiquetées sur deux objets distincts

L'instruction `b=2` requiert une modification du contenu étiqueté par `b`. Un entier est un type *non mutable*, son contenu n'est pas directement modifiable. La variable `b` est alors étiquetée sur un nouvel objet qui est l'entier 2. La variable `a` reste étiquetée sur le même objet.

On peut créer une variable vide sans lui affecter de valeur avec l'objet `None`.

```
>>> a=None;a
>>>
```

Hormis cette dernière situation, une variable qui n'a pas été initialisée n'existe pas.

```
>>> x
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

2 Affectation

Proposition 8. *En python, le typage est* dynamique *: Le type d'une variable est déterminé par le type de l'expression qui lui est affectée.*

Expérimentation :

```
>>> a=1
>>> type(a)
<class 'int'>
```

Python crée la variable `a` et choisit son type en fonction de l'expression saisie (l'utilisateur ne déclare pas le type comme ce peut être le cas dans d'autres langages). Le type d'une variable n'est pas figé, il peut évoluer avec une nouvelle assignation de la variable.

```
>>> a=1.
>>> type(a)
<class 'float'>
```

Problème classique : Échanger deux variables

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

Les instructions qui précèdent ne réalisent pas la permutation des variables `a` et `b`. En effet, l'instruction `b=a` fait que `b` et `a` sont étiquetés sur le même objet. L'instruction qui suit `a=b` est sans effet et la valeur de `a` ne se voit pas assigner la valeur initiale de `b` d'où le résultat observé. La démarche habituelle pour résoudre ce problème consiste à utiliser une variable intermédiaire.

```
>>> a=1
>>> b=2
>>> c=a
>>> a=b
>>> b=c
>>> a
2
>>> b
1
```

On verra dans la section suivante comment réaliser une telle permutation sans variable intermédiaire à l'aide de types composés et d'une manière plus simple que celle proposée ci-avant.

3 Opérations

Proposition 9. *Le langage python permet de réaliser simultanément affectations et opérations sur des variables numériques au moyen des instruction* `+=`, `-=`, `*=` *et* `/=`.

Expérimentation :

```
>>> a=1
>>> a+=1
>>> a
2
>>> a*=3
>>> a
6
>>> a/=2
>>> a
3.0
```

L'instruction `a+=1` équivaut à l'instruction `a=a+1` et de même pour les autres instructions.

Proposition 10. *La fonction* `print` *permet l'affichage de la valeur d'une variable.*

Remarque : Dans un usage avec console, ces instructions ne sont pas très utiles mais elles sont incontournables dans l'écriture des programmes.

Exercice : Prédire le résultat des instructions suivantes :

```
>>> a=3
>>> a*=a
>>> b=2
>>> b+=a
>>> a
>>> b
```

Corrigé :

```
>>> a
9
>>> b
11
```

III Types composés

1 Les tuples

Définition 7. *On peut définir en python des* tuples *qui sont une succession non modifiable finie et ordonnée d'objets rassemblés au sein d'un seul.*

Expérimentation :

```
>>> a=(4,5,6)
>>> a
(4, 5, 6)
>>> type(a)
<class 'tuple'>
>>> a=4,5,6
>>> a
(4, 5, 6)
```

On remarque que les parenthèses sont superflues. Créer un `tuple` vide est intuitif mais créer un `tuple` à un élément l'est un peu moins.

```
>>> a=()
>>> type(a)
<class 'tuple'>
>>> a=(1,)
>>> a
(1,)
>>> a=1,
>>> a
(1,)
>>> type(a)
<class 'tuple'>
```

On peut créer un tuple par répétition d'un motif.

```
>>> (1,)*5
(1, 1, 1, 1, 1)
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

On accède aux différentes composantes d'un tuple avec les crochets, l'indexation commençant à zéro.

```
>>> a=4,5,6
>>> a[1]
5
>>> a[0]
4
>>> a[-1]
6
```

En revanche, on ne peut pas modifier une composante d'un tuple. C'est un type non mutable.

```
>>> a=4,5,6
>>> a[0]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

On peut extraire un sous tuple d'un autre avec du *slicing*, en spécifiant une plage d'indices entre crochets `[i:j]` de `i` à `j` exclu (si `i` est omis, on extrait depuis le début et si `j` est omis, on extrait jusqu'à la fin). L'accès aux composantes en commençant par la fin se fait avec une indexation négative : `[-1]` pour la dernière composante, `[-2]` pour l'avant-dernière, ...

```
>>> a=4,5,6
>>> a[1:2]
(5,)
>>> a[:2]
(4,5)
>>> a[1:]
(5, 6)
```

On peut aussi renvoyer un tuple renversé avec du slicing :

```
>>> a=(4,5,6)
>>> a[::-1]
(6, 5, 4)
```

Proposition 11. *On peut en extraire un sous-tuple d'un tuple par slicing selon les règles suivantes :*

- `[i:j]` *: de* `i` *à* `j` *exclu par pas de* 1 *;*
- `[i:]` *: de* `i` *à la fin par pas de* 1 *;*
- `[:j]` *: du début à* `j` *exclu par pas de* 1 *;*
- `[i:j:h]` *: de* `i` *à* `j` *exclu par pas de* `h` *;*
- `[i::h]` *: de* `i` *à la fin par pas de* `h` *;*
- `[:j:h]` *: du début à* `j` *exclu par pas de* `h` *;*
- `[::h]` *: du début à la fin par pas de* `h` *;*
- `[::-1]` *: de la fin au début par pas de* 1.

On peut construire des tuples composites avec différents type d'objets. Un tuple peut notamment contenir d'autres tuples. On accède alors aux composantes par une indexation multiple.

```
>>> a=(1,1.,True)
>>> a
(1, 1.0, True)
>>> a=((1,2),((3,4),5))
>>> a
((1, 2), ((3, 4), 5))
>>> a[1]
((3, 4), 5)
>>> a[1][0]
(3, 4)
>>> a[1][0][1]
4
```

Proposition 12. *En langage python, on utilise les instructions suivantes sur les* `tuple` *:*

- *l'instruction* `len` *renvoie la taille d'un* `tuple`*,*
- *l'instruction* `+` *permet de concaténer des* `tuple`*,*
- *l'instruction* `in` *teste l'appartenance d'un élément à un* `tuple`*.*

Expérimentation :

```
>>> a=(2,4,6,4,1)
>>> len(a)
5
>>> 7 in a
False
>>> a+(3,6)
(2, 4, 6, 4, 1, 3, 6)
```

À l'instar de ce qui existe pour les formats de de nombres, on peut concaténer et affecter simultanément avec l'instruction `+=`. Le `tuple` vide se note `()`.

```
>>> a=()
>>> a+=(1,2,3)
>>> a
(1, 2, 3)
```

On peut utiliser des tuple pour réaliser des affectations simultanées.

```
>>> x,y=4,5
>>> x
4
>>> y
5
```

Dans l'instruction `x,y=4,5`, le coupe `4,5` est un tuple tandis que le couple de variables `x,y` qui ne sont pas encore créées forme une *target list*.

Exercice : En utilisant des `tuple`, réaliser un échange entre deux variables puis une permutation cyclique entre trois variables sans créer de variables intermédiaires.

Corrigé :

```
>>> a,b=1,2
>>> a,b=b,a
>>> a,b
(2, 1)
>>> a,b,c=1,2,3
>>> a,b,c=b,c,a
>>> a,b,c
(2, 3, 1)
```

Exercice : Soient $a = 1$ et $b = 2$. Réaliser sans variable intermédiaire l'opération suivante :

$$(a, b) \leftarrow \left(\frac{a+b}{2}, \sqrt{ab}\right)$$

Corrigé :

```
>>> a,b=1,2
>>> a,b=(a+b)/2,np.sqrt(a*b)
>>> a,b
(1.5, 1.4142135623730951)
```

2 Les chaînes de caractères

Définition 8. *En python, les* chaînes de caractères *ou* string *en anglais, objets de type* `str`*, servent à coder du texte. Les textes composés d'un seul symbole sont appelés* caractères.

Expérimentation :

```
>>> x='bonjour'
>>> type(x)
<class 'str'>
>>> x="python c'est bien!"
>>> type(x)
<class 'str'>
```

On peut créer une chaîne de caractères par répétition d'un motif.

```
>>> "1"*5
'11111'
>>> "12"*5
'1212121212'
```

Contrairement aux tuples, une chaîne de caractères est homogène : elle n'est composée que de caractères. La saisie d'une chaîne de caractères peut se faire indifféremment entre apostrophes ou entre guillemets.

Proposition 13. *On peut extraire une sous-chaîne d'une chaîne par slicing selon les règles suivantes :*

- `[i:j]` *: de* `i` *à* `j` *exclu par pas de* 1 *;*
- `[i:]` *: de* `i` *à la fin par pas de* 1 *;*
- `[:j]` *: du début à* `j` *exclu par pas de* 1 *;*
- `[i:j:h]` *: de* `i` *à* `j` *exclu par pas de* `h` *;*
- `[i::h]` *: de* `i` *à la fin par pas de* `h` *;*
- `[:j:h]` *: du début à* `j` *exclu par pas de* `h` *;*
- `[::h]` *: du début à la fin par pas de* `h` *;*
- `[::-1]` *: de la fin au début par pas de* 1*.*

Comme pour les tuples, les composantes ne sont pas modifiables, c'est un type non mutable.

```
>>> x='bonjour'
>>> x[3:7]
'jour'
>>> x[0]='t'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On dispose des mêmes instructions que pour les tuples plus d'autres spécifiques aux chaînes.

Proposition 14. *En langage python, on utilise les instructions suivantes sur les* `str` *:*
- *l'instruction* `len` *renvoie la taille d'un* `str`*,*
- *l'instruction* `+` *permet de concaténer des* `str`*,*
- *l'instruction* `in` *teste l'appartenance d'un élément à un* `str`*,*

Une chaîne vide se crée avec '' ou "" et peut se concaténer avec `+=`.

```
>>> x='bon';x+='jour'
>>> x
'bonjour'
```

La conversion d'une chaîne vers un type simple est possible quand le contenu s'y prête.

```
>>> int('123')
123
>>> str(123)
'123'
>>> float("1.2")
1.2
>>> str(1.2)
'1.2'
```

Exercice : Tester si la séquence 135 est présente dans l'écriture décimale par défaut de π et de $\sqrt{2}$.

Corrigé :

```
>>> '135' in str(np.pi)
False
>>> np.pi
3.141592653589793
>>> '135'in str(np.sqrt(2))
True
>>> np.sqrt(2)
1.4142135623730951
```

3 Les listes

Définition 9. *On peut définir en python des listes modifiables composés d'objets rassemblés au sein d'un seul. Le format sous python de ces listes est appelé* `list`*.*

⚠ Les listes sont modifiables : on dit qu'elles sont *mutables*. Cette propriété est fondamentale.

Expérimentation :

```
>>> a=[4,5,6]
>>> type(a)
<class 'list'>
>>> a=[]
>>> type(a)
<class 'list'>
```

On peut créer une liste par répétition d'un motif.

```
>>> [1]*5
[1, 1, 1, 1, 1]
>>> [1,2]*5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Proposition 15. *On peut en extraire une sous-liste d'une liste par slicing selon les règles suivantes :*

- `[i:j]` *: de* `i` *à* `j` *exclu par pas de* 1 *;*
- `[i:]` *: de* `i` *à la fin par pas de* 1 *;*
- `[:j]` *: du début à* `j` *exclu par pas de* 1 *;*
- `[i:j:h]` *: de* `i` *à* `j` *exclu par pas de* `h` *;*
- `[i::h]` *: de* `i` *à la fin par pas de* `h` *;*
- `[:j:h]` *: du début à* `j` *exclu par pas de* `h` *;*
- `[::h]` *: du début à la fin par pas de* `h` *;*
- `[::-1]` *: de la fin au début par pas de* 1.

Comme pour les `tuples` et les chaînes, une indexation négative permet d'accéder aux composantes en commençant par la fin.

```
>>> a=[5,3,9,1,3,4]
>>> a[0]
5
>>> a[1:4]
[3, 9, 1]
>>> a[2:]
[9, 1, 3, 4]
>>> a[0]=6
>>> a
[6, 3, 9, 1, 3, 4]
>>> a[-1]
4
```

Comme pour les tuples et les chaînes, on peut aussi renverser une liste par slicing.

```
>>> a=[5,3,9,1,3,4]
>>> a[::-1]
[4, 3, 1, 9, 3, 5]
```

On peut utiliser le slicing pour des affectations par tranche.

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

L'affectation `a[5:]=[0]*2` a aussi eu pour effet de raccourcir la liste initiale.

Exercice : Soit `a` une liste de taille paire. Échanger dans la variable `a` les termes d'indices pairs et impairs.

Corrigé : L'instruction `a[::2],a[1::2]=a[1::2],a[::2]` réalise l'échange attendu. Par exemple, on saisit :

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

Le mécanisme d'affectation d'une variable `list` soulève des difficultés inédites par rapport aux autres types.

```
>>> a=[1,2,3]
>>> b=a
>>> a[0]=5
>>> a
[5, 2, 3]
>>> b
[5, 2, 3]
>>> id(a)
58292232
>>> id(b)
58292232
```

L'instruction `b=a` crée une nouvelle variable étiquetée sur le même objet que celui étiqueté par `a`. Or, une liste est un objet mutable. Ainsi, la modification d'une composante de `a` modifie l'objet étiqueté par `a` et comme `b` est étiquetée sur le même objet, on a le même résultat par `a` ou `b`. L'affichage des identifiants confirme ce fait.
Pour que la variable `b` soit une liste indépendante de `a` et de même valeur, on utilise l'instruction `list` ou l'indexation globale `[:]`.

```
>>> a=[1,2,3]
>>> b=list(a)
>>> c=a[:]
>>> b
[1, 2, 3]
>>> b[1]=5
>>> b
[1, 5, 3]
>>> a
[1, 2, 3]
>>> c
[1, 2, 3]
```

On peut construire des listes composites avec différents type d'objets. Une liste peut notamment contenir d'autres listes.

```
>>> a=[1,2,'bonjour',(1,2,3),[[1],[2,3]]]
>>> a
[1, 2, 'bonjour', (1, 2, 3), [[1], [2, 3]]]
>>> a[4]
[[1], [2, 3]]
>>> a[4][1]
[2, 3]
```

Pour une liste contenant des sous-listes, les mêmes précautions liées au caractère mutable sont à prendre. Pour réaliser une copie de liste complètement indépendante, il faut effectuer une copie en profondeur :

```
>>> from copy import deepcopy
>>> a=[[1],2]
>>> b=deepcopy(a)
>>> b[0][0]=3
>>> b
[[3], 2]
>>> a
[[1], 2]
```

Proposition 16. *En langage python, on utilise les instructions suivantes sur les listes :*
- *l'instruction* `len` *renvoie la taille d'une liste,*
- *l'instruction* `+` *permet de concaténer des listes,*
- *l'instruction* `in` *teste l'appartenance d'un élément à une liste,*

Proposition 17. *En langage python, on utilise les méthodes suivantes sur les listes :*
- *la méthode* `append` *ajoute un élément en fin de liste,*
- *la méthode* `pop` *renvoie et supprime le dernier élément d'une liste,*
- *la méthode* `sort` *ordonne une liste par ordre croissant.*

Remarque : Les méthodes sont des fonctions conçues pour une classe d'objets. Quand on saisit le nom d'un objet `list` suivi d'un point puis une tabulation, une liste déroulante apparaît proposant l'ensemble des méthodes dédiées à cet objet.

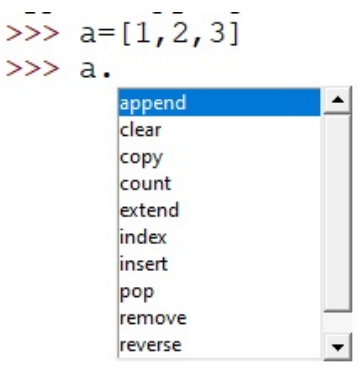

FIGURE 5 – Méthodes d'un objet `list`

Expérimentation :

```
>>> a=[4,2,3]
>>> a+=[5,6,5]
>>> a
[4, 2, 3, 5, 6, 5]
>>> a.append(1)
>>>
>>> a
[4, 2, 3, 5, 6, 5, 1]
>>> a.pop()
1
>>> a
[4, 2, 3, 5, 6, 5]
>>> a.sort()
>>> a
[2, 3, 4, 5, 5, 6]
```

On peut utiliser des `list` pour réaliser des affectations simultanées.

```
>>> x,y=[4,'bonjour']
>>> x
4
>>> y
'bonjour'
```

Des conversions entre les types composés sont possibles avec les instructions `tuple`, `str` et `list`. La conversion d'une chaîne en liste ou `tuple` n'est pas réversible.

```
>>> list((1,2,3))
[1, 2, 3]
>>> tuple([1,2,3])
(1, 2, 3)
>>> list('123')
['1', '2', '3']
>>> str(list('123'))
"['1', '2', '3']"
```

Exercice : Soient `a` et `b` des chaînes de caractère. Écrire des instructions permettant de tester si les chaînes sont des anagrammes.

Corrigé : On convertit les chaînes en liste que l'on peut ensuite ordonner avec la méthode `sort`.

```
>>> a="bonjour"
>>> b="jourbno"
>>> ta=list(a);ta
['b', 'o', 'n', 'j', 'o', 'u', 'r']
>>> ta.sort();ta
'b', 'j', 'n', 'o', 'o', 'r', 'u']
>>> tb=list(b);tb
['j', 'o', 'u', 'r', 'b', 'n', 'o']
>>> tb.sort();tb
['b', 'j', 'n', 'o', 'o', 'r', 'u']
>>> ta==tb
True
```

Exercice : Construire efficacement les listes suivantes :

1. `[1, 1, 1, ... 1]` de taille 10,
2. `[[1,1,1,1],[0,0,0]]`,
3. `[[1,0,1,0,1,0,1],[1,1,1,1,1,1,1]]`.

Corrigé : On saisit :

```
>>> [1]*10
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> [[1]*4,[0]*3]
[[1, 1, 1, 1], [0, 0, 0]]
>>> [[1,0]*3+[1],[1]*7]
[[1, 0, 1, 0, 1, 0, 1], [1, 1, 1, 1, 1, 1, 1]]
```

Exercice : L'instruction `[[1]*3]*2` fournit la liste `[[1,1,1],[1,1,1]]`. Critiquer le procédé et proposer une démarche alternative.

Corrigé : On saisit :

```
>>> a=[[1]*3]*2
>>> a
[[1, 1, 1], [1, 1, 1]]
```

En apparence, tout va bien. Le défaut de ce procédé est qu'il y a duplication de listes et donc en réalité duplication d'étiquettes sur une même liste : les sous-listes `a[0]` et `a[1]` ont la même référence et une modification du contenu de `a[0]` entraine également une modification de `a[1]`.

```
>>> a[0][1]=2
>>> a
[[1, 2, 1], [1, 2, 1]]
```

Pour contourner ce problème, on peut utiliser les constructions alternatives suivantes :

```
>>> a=[[1]*3,[1]*3]
>>> a
[[1, 1, 1], [1, 1, 1]]
>>> a[0][1]=2
>>> a
[[1, 2, 1], [1, 1, 1]]
```

ou aussi

```
>>> a=[]
>>> a.append([1]*3)
>>> a.append([1]*3)
>>> a
[[1, 1, 1], [1, 1, 1]]
>>> a[0][1]=2
>>> a
[[1, 2, 1], [1, 1, 1]]
```

4 Les ranges

Définition 10. *On peut définir en python des séquences qui suivent une progression arithmétique : les* `range`.

Si l'on saisit `range(n)`, on crée une séquence de 0 à $n-1$. Si l'on saisit `range(deb,fin)`, on crée une séquence de `deb` à `fin-1` par pas de 1. Si l'on saisit `range(deb,fin, step)`, on crée une séquence de `deb` à `fin` exclu par pas de `step`. L'instruction `help(range)` fournit un descriptif de la fonction.

La conversion en liste permet de voir le contenu d'un range.

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

```
>>> list(range(2,20,2))
[2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

C'est un type non mutable.

IV Synthèse

1 Les différents types

type	nature	simple/composé	homogène/hétérogène	mutable
`int`	entier	simple	-	-
`bool`	booléen	simple	-	-
`float`	flottant	simple	-	-
`complex`	complexe	simple	-	-
`str`	chaîne de caractères	composé	homogène	non
`tuple`	n-uplet	composé	hétérogène	non
`list`	liste	composé	hétérogène	oui
`range`	séquence arithmétique	composé	homogène	non

Le nom du type est également une instruction pour convertir en le type concerné (sous réserve de compatibilité). Ainsi, l'instruction `str(123)` renvoie `"123"` et l'instruction `int("123")` renvoie `123`.

2 Opérations, affectations, tests

Opérations

instruction	action	type(s) concerné(s)
`+,-,*,/, **`	opérations arithmétiques	`int`, `bool`, `float`, `complex`
`%, //`	reste, quotient	`int`
`abs`	module	`int`, `bool`, `float`, `complex`
`not, and, or`	opérations logiques	`bool`
`+`	concaténation	`tuple`, `str`, `list`
`*`	répétition de motif	`tuple`, `str`, `list`
`len`	taille	`tuple`, `str`, `list`

Opérations/affectations

instruction	action	type(s) concerné(s)
`+=, -=, ...`	opérations arithmétiques et affectations	`int`, `bool`, `float`, `complex`
`%=, //=`	reste, quotient et affectation	`int`
`+=`	concaténation et affectation	`tuple`, `str`, `list`
`*=`	répétition de motif et affectation	`tuple`, `str`, `list`

Pour une liste, la méthode `append` permet d'ajouter un élément en fin de liste et la méthode `pop` renvoie et supprime le dernier élément de la liste.

Tests

test	action	type(s) concerné(s)
`==`	test d'égalité	tous
`!=`	test de différence	tous
`<,>,<=,>=`	relation d'ordre	`int`, `float`
`in`	test d'appartenance	`tuple`, `str`, `list`

Copies indépendantes

Pour réaliser une copie indépendante de la variable `a` dans la variable `b` :

instruction	type(s) concerné(s)
`b=a`	tous sauf `list`
`b=a[:]` ou `b=list(a)`	`list` sans sous-listes
`b=deepcopy(a)`	`list` (avec ou sans sous-listes)

3 Règles de slicing

Pour l'extraction (et aussi l'affectation pour les listes) des types composés, les règles sont décrites par :

syntaxe	action
`[i:j]`	de `i` à `j` exclu par pas de 1
`[i:]`	de `i` à la fin par pas de 1
`[:j]`	du début à `j` exclu par pas de 1
`[i:j:h]`	de `i` à `j` exclu par pas de `h`
`[i::h]`	de `i` à la fin par pas de `h`
`[:j:h]`	du début à `j` exclu par pas de `h`
`[::h]`	du début à la fin par pas de `h`
`[::-1]`	de la fin au début par pas de 1

L'accès aux composantes en commençant par la fin se fait avec une indexation négative : `[-1]` pour la dernière composante, `[-2]` pour l'avant-dernière, ...