1 Environnement de programmation

2 Variables, Types

2.1 Quelques définitions

Définition Valeurs – Les valeurs désignent les données manipulées par un algorithme ou une fonction. Une valeur peut ainsi être : un nombre, un caractère, une chaîne de caractères, une valeur de vérité (`Vrai` ou `Faux`) *etc.*

En Python, comme dans la plupart des langages de programmation, ces valeurs sont **typées** selon l'objet qu'elles représentent : il y a ainsi des valeurs de **type** entier, flottant, chaîne de caractères, booléens ... Leur représentation en mémoire varie beaucoup d'un langage à l'autre, mais ce sont souvent les mêmes objets que l'on cherche à traduire.

Définition Expression – Une expression est une suite de caractères faisant intervenir des valeurs et des opérations, afin d'obtenir une nouvelle valeur. Pour calculer cette nouvelle valeur, la machine doit évaluer l'expression. Voici des exemples d'expressions : `42 , 1+4 , 1.2 / 3.0 , x+3`.

En Python, pour évaluer une expression, il suffit de la saisir dans un interpréteur (consoles), qui calcule et affiche alors la valeur qu'il a calculée.

```
>>> 42
       42
>>> 1+4
       5
```

Définition Variable – Une **variable** désigne une zone mémoire de l'ordinateur dans la RAM. Il s'agit d'un endroit où l'on peut **stocker** une valeur, y **accéder** et **changer** cette valeur.

Pour faire référence à une variable, on utilise un nom de variable, en général composé d'une ou plusieurs lettres. Dans la mesure du possible, on choisit un nom explicite, ce qui permet une meilleure lecture du programme.

2.2 Types simples

En programmation, associer un type à une valeur permet :

- de classer les valeurs en catégories similaires. Par exemple, tous les entiers (type int), tous les flottants (type float)...
- de contrôler quelles opérations sont faisables sur ces valeurs : par exemple, tout entier (type int) pourra être utilisé dans une soustraction, ..., alors qu'une chaîne ne le pourra pas. Il sera également impossible d'additioner un entier et un booléen.

Une expression n'a a priori pas de type, car le type de la valeur qu'elle renvoie dépend des types des sous-expressions. Ainsi `a+b` sera un entier (resp. un flottant, une chaîne) si `a` et `b` le sont, mais sera un flottant si `a` est un entier et `b` un flottant.

Pour afficher le type d'une valeur ou d'une expression après l'avoir évaluée, on utilise `type`.

```
>>> type(42)
       int
>>> type(1.2/3.0)
       float
```

2.2.1 Entiers

Définition Entiers – Ce type est noté `int` en Python .

Constantes : ce sont les entiers relatifs écrits en base 10. Ils ne sont pas bornés en Python .

- **Opérateurs :**
  1. $+$: addition usuelle ;
  2. $-$: soustraction usuelle (`15-9` renvoie 6), mais aussi opposé (`-4`) ;
  3. `//` : division entière : `a//b` correspond au quotient de la division euclidienne de `a` par **b** si `b` est strictement positif (`256 // 3` renvoie 85 car $256 = 85*3+1$). Mais si `b` est strictement négatif, alors `a//b` renvoie ce quotient moins 1 ($15 = (-4)\times(-4) - 1 = (-4)\times(-3)+3$, le quotient de la division euclidienne de 15 par -4 est -3, mais `15//-4` renvoie `-4`). Cette division n'est pas définie si `b` est nul ;
  4. `%` : modulo : même remarque que dans le point précédent, relativement au reste de la division euclidienne cette fois (`256 % 3` renvoie `1`, `15%-4` renvoie `-1`).
  5. `**` : exponentiation (`2**3` renvoie 8).

Les règles de précédence, autrement dit les règles de priorité entre opérations, sont similaires aux règles mathématiques usuelles.

2.2.2 Flottants

Définition Flottants – Ce type est noté `float` en Python .

Constantes : ce sont les nombres à virgule flottante. Nous en donnerons une définition précise dans un chapitre ultérieur : pour simplifier, disons pour l'instant que ce sont des nombres à virgule, avec un nombre borné de chiffres dans leur écriture.

- **Opérateurs :**
  1. + : addition usuelle ;
  2. − : soustraction usuelle, et aussi opposé.
  3. / : division usuelle.
  4. ** : exponentiation.

2.2.3 Booléens

Définition Expression – Booléen Ce type est noté `bool` en Python.

Constantes : il y en a deux : `True` et `False`.

- **Opérateurs :**
  1. `not` : négation usuelle ;
  2. `and` : conjonction usuelle ;
  3. `or` : disjonction usuelle.
- **Opérateurs de comparaison :**
  1. `==` : test d'égalité : `2==3` renvoie `False`, `4==4` renvoie `True`.
  2. `!=` : `a != b` est une écriture équivalente à `not (a == b)`.
  3. `<,>,<=,>=` : ce à quoi on s'attend.

▪ **Exemple** `True or False` renvoie `True`. `not(False or True) and True` renvoie `False`. ▪

R Python permet les chaînes de comparaisons : `1<2<3` renvoie `True`, et `(1<2<-2) and (-5<2<6)` renvoie `False`.

2.3 Conversions – *Cast*

Il est possible de convertir en entier en flottant avec la fonction `float`. La réciproque est possible dans une certaine mesure : `float(2)` renvoie `2.0`, `int(2.0)` renvoie 2, mais `int(2.5)` renvoie 2 et `int(-3.5)` renvoie `-3`.

La fonction `int` appliquée à un nombre flottant procède donc à une troncature, ce n'est pas la fonction donnant la partie entière d'un nombre flottant (fonction `floor` du module `math`).

2.4 Types composés

Définition On dit qu'une valeur est de type composé si elle est formée à partir de plusieurs autres valeurs de types plus simples. De nombreuses constructions sont définies sur tous les types composés.

2.4.1 n-uplets ou tuples

Définition Tuple – Ce type est noté `tuple` en Python .

Construction et composantes : les n-uplets généralisent les couples. Un couple s'écrira `(a,b)`, un triplet `(a,b,c)` ... etc. Il existe même des 1-uplets : il faut les écrire `(a,)`.¨

Il n'est pas obligatoire que les éléments d'un tuple soient de même type. Par exemple, `(1,1.2,True)` est un triplet tout à fait valable.

On accède à la k^e coordonnée d'un tuple `t` par la commande `t[k]`. Attention : **en Python , les coordonnées sont numérotées à partir de 0** ! On dit que k est l'indice de `t[k]`. Ainsi, si `t = (1,2,3)`, `t[0]` vaut 1, `t[2]` vaut 3, et `t[3]` n'existe pas.

On dit que les tuples sont **immuables** ou **non mutables** : il n'est pas possible de changer un élément d'un tuple.

- **Opérateurs :**
  1. + : concaténation. `(1,2,3)+(1,4,5,6)` renvoie `(1,2,3,1,4,5,6)` ;
  2. `len` : longueur du tuple. `len(1,2,3)` renvoie 3 ;
  3. `in` : test d'appartenance. `3 in (1,2,3)` renvoie `True`, alors que `3 in (1,2,4,5)` renvoie `False` ;
  4. le slicing, qui permet de construire des sous-tuples : `(5,2,6,7,8,9)[1:5]` renvoie `(2,6,7,8)` (attention à la borne de droite !).

2.4.2 Chaînes de caractères

Définition Chaînes de caractères – Ce type est noté `str` en Python , pour string en anglais.

Construction et composantes : ce sont des suites finies de caractères, un caractère étant en python un caractère du clavier. On les note entre guillemets ou apostrophes : ''`Ceci est un chaîne de caractères`'' ou '`Ceci est une chaîne de caractères`'. La chaîne vide se note "" ou ''. Un caractère est une chaîne de longueur 1.

On accède aux composantes d'une chaîne de la même manière que pour un tuple.

- **Opérateurs :** ce sont les mêmes que pour les tuples.

2.5 Listes

Définition Listes – Ce type est noté `list` en Python .

Comme les tuples, les listes sont des suites finies de valeurs arbitraires, mais cette fois ils sont **mutables** : on peut changer la valeur d'une composante d'une liste.

Les listes s'écrivent entre crochets. On change la valeur de la k^e composante d'une liste `L` grâce à la commande `L[k] = n`, où `n` est la nouvelle valeur. Ainsi :

```
L = [1,2,3]
L[0] = 4
L
```

Les autres opérateurs sont les mêmes que pour les tuples et les chaînes. Nous reviendrons plus tard sur les listes.

2.6 Conversions

On peut convertir des types simples en chaînes avec la fonction `str` : `str(2.3)` renvoie `'2.3'`.
À l'inverse, `int, float` et `bool` permettent de faire l'inverse quand cela est cohérent : **bool('True')** renvoie `True`, mais `int('2.3')` renvoie une erreur.

Il est également possible de passer d'un type composé à un autre grâce aux fonctions `str`, `tuple` et `list` : `str((1,'a',[1,2]))` renvoie `"(1,'a',[1,2])"`, `tuple([1,2,3] )` renvoie `(1,2,3)` et `list('Coucou ?')` renvoie `['C', 'o', 'u', 'c', 'o', 'u', ' ', '?']`.

2.7 Affectation

Quand on stocke une valeur `d` dans une variable `var`, on dit que l'on **affecte** `d` à la variable `var`. La valeur `d` est encore appelée **la donnée**.

En Python , cette affectation est faite avec la commande `=`, comme suit.

```
>>> var = 1
```

Les variables (et donc l'affectation) sont incontournables : si vous ne stockez pas une donnée (résultat par exemple d'un calcul), vous ne pourrez pas la réutiliser, ni la modifier.

```
>>> x = 42 # x prend la valeur 42.
>>> x = x + 2 # on additionne àla donnée stockée dans x le nombre 2 et on
              # stocke le résultat de nouveau dans x
>>> x # on accède àla valeur qui est mémorisée dans x.
       44
```

Dans la variable affectée, la nouvelle donnée « écrase » la donnée précédente : on perd son ancienne valeur.

Dans un programme Python, on utilisera l'instruction `print(x)` pour afficher dans la console la valeur de la variable `x`.

Il est également possible d'affecter des valeurs à plusieurs variables en une fois, en utilisant des tuples. Ainsi : `a,b = (1,2)` affecte la valeur 1 à `a` et la valeur 2 à `b`.

R En Python, le **typage** des variables est **dynamique**. Cela signifie que lorsqu'on affecte une valeur à une variable, Python reconnaît automatiquement le type de la valeur. Ce n'est pas le cas de tous les langages de programmation.

Notion d'adressage

En Python, le passage des variables se fait par **référence**. C'est-à-dire que lorsqu'on manipule une variable (qu'on l'envoie à une fonction par exemple), on ne donne pas comme argument la valeur de la fonction, mais son adresse mémoire. Pour les types non mutables (entiers, flottants, chaîne de caractères, tuples), le passage par référence ne pose pas de problèmes. Pour les types mutables (listes), le passage par référence peut conduire à des résultats « inattendus » notamment lors de la copie de listes.

```
>>> a=2
>>> print(a)
       2
>>> a=3
>>> print(a)
       3
>>> a=[1,2]
```

```
>>> print(a)
       [1,2]
>>> b=a
>>> a[0]=2
>>> print(a,b)
       [2,2] [2,2]
```

Ici l'affectation `b=a` ne crée pas une nouvelle liste `b`. On crée juste une nouvelle variable `b` qui a la même adresse mémoire que `a`. Ainsi en modifiant `a`, on modifie `b`.

3 Fonctions

3.1 Objectif : modularité des programmes

Pour répondre à un problème donné, il est souvent nécessaire d'enchaîner plusieurs voire un nombre important d'instructions. Pour améliorer la lisibilité d'un programme et aussi pour pouvoir réutiliser ces instructions classiques (dans le même programme ou dans un autre), on privilégie les programmes simples (ou sous-programmes) appelés **fonctions**.

Dans chaque langage, il y a déjà un nombre incalculable de fonctions déjà construites (en Python, par exemple, `print`) mais on s'aperçoit très vite que l'on a besoin de créer ses propres fonctions.

- Pour bien se faire comprendre, il est important de choisir un nom de fonction explicite. Il faut aussi mettre des commentaires pour expliquer ce que fait la fonction.
- Il est nécessaire de bien cibler les paramètres dont on a besoin c'est-à-dire les données nécessaires à l'exécution de la fonction.
- Il faut repérer ce que renvoie la fonction : rien (exemple : un simple affichage ou la modification d'un fichier...), un nombre entier, un flottant, une liste...
- Enfin, il faut toujours tester sa fonction sur un ensemble significatif de valeurs pour repérer d'éventuelles erreurs ou manquements.

3.2 Écriture en Python

```
def nom_de_la_fonction(parametres):
    """ commentaire expliquant la fonction """
    #bloc d'instructions
    return(resultat)
```

En Python , l'indentation est signifiante : après le mot-clé `def`, chaque ligne indentée fait partie de la fonction. La première ligne non indentée rencontrée marque la fin de la fonction : cette ligne ne fait plus partie de la fonction, mais celles qui précède en font partie. Il est donc impératif d'indenter quand il le faut, et seulement quand il le faut.

Annotations des fonctions

Afin de préciser à l'utilisateur ou au lecteur de votre programmes quels sont les types d'arguments ou le type retourné par la fonction, il est possible d'utiliser des annotations. Celles-ci ont un rôle uniquement documentaire.

```
def foo(par: int) -> float : # Ces annotations indiquent que la fonction prend
                             # comme un argument un entier
                             # et renvoie un float.
    """ commentaire expliquant la fonction """
    #bloc d'instructions
    return(2.*par)
```

4 Structures algorithmiques

4.1 L'instruction conditionnelle

4.1.1 Algorithme

Quand on veut écrire un programme, on souhaite établir des connections logiques entre les instructions. Ainsi, l'instruction conditionnelle a pour objet d'intervenir dans le choix de l'instruction suivante en fonction d'une expression booléenne qu'on désignera par **condition** :

Si condition
alors bloc d'instructions 1
sinon bloc d'instructions 2
Fin-du-Si

signifie que

- **Si** la condition est vérifiée (expression booléenne=`True`)
  alors le programme exécute les instructions du bloc 1 ;
- **si** la condition n'est pas vérifiée (expression booléenne=`False`)
  alors le programme exécute les instructions du bloc 2.

4.1.2 Syntaxe en Python

```
if condition :
       bloc d instructions 1
else :
       bloc d instructions 2
```

- **Si** et **Sinon** se traduisent par `if` et `else`.
- **Alors** se traduit par « **:** » en bout de ligne et une indentation de toutes les lignes du bloc 1.
- **Fin-du-Si** se traduit par un retour à la ligne sans indentation.

4.1.3 Exemple

On veut tester si un nombre x est proche de 3 à 10^{-4} près. On peut alors écrire la fonction suivante.

```
def est_proche(x):
    """x est proche de 3 à10**-4 près ?"""
    distance = abs(x-3)
    if distance <= 10**(-4) :
        return True
    else :
        return False
```

R La partie **sinon** est optionnelle. Sans elle, si la condition n'est pas vérifiée, alors la machine n'exécute rien.

R On pouvait très bien remplacer cette boucle conditionnelle avec un usage astucieux des booléens, comme suit.

```
def est_proche(x):
    """x est proche de 3 à10**-4 près ?"""
    distance = abs(x-3)
    return distance <= 10**(-4)
```

La fonction est alors plus concise, mais plus difficilement lisible.

4.1.4 À propos des conditions

L'expression booléenne derrière le **si** joue le rôle de test. Pour exprimer cette condition, on a besoin des **opérateurs de comparaison** (inférieur strict, supérieur strict, inférieur ou égal, supérieur ou égal, égal à, différent de) et des **connecteurs logiques** (non, et, ou).

- Calcul du carré d'un nombre positif.

```
>>> x=4
>>> if x >= 0 :
        car = x**2
```

```
>>> car
       16
```

- Condition avec un « et ».

```
x = 0.5
if x >= -1 and x <= 1 :
    print("Il existe un angle theta tel que x = cos(theta).")
```

4.1.5 Imbrication de plusieurs conditions

On peut se trouver face à un problème qui se scinde en plus de deux cas (par exemple, dans le cas des équations du second degré, on teste si le discriminant est strictement positif, nul ou strictement négatif). Dans ce cas, voici comment procéder.

```
 if condition 1 :
       bloc d instructions 1
 elif condition 2 :
       bloc d instructions 2
 elif condition 3 :
       bloc d instructions 3
.
.
.
 else :
       bloc final
```

- **Sinon si** se traduit par `elif`.

▪ **Exemple** Écrire les deux séquences d'instructions de la remarque **??** en Python, et constater que l'on obtient bien deux résultats différents. ▪

Voici une fonction qui calcule le maximum de trois entiers `a, b, c` :

```
def max3 (a, b, c) :
    """ renvoie le maximum de a, b ,c.
    précondition : a, ,b et c sont 3 entiers """
    if a <= c and b <= c :
        return c
    elif a <= b and c < b :
        return b
    else :
        return a
```

4.2 Boucles définies

	Pour variable **dans** liste **répeter** bloc d'instructions b **Fin-de-la-boucle**
	signifie que
pour chaque élément de la liste `liste`, le programme exécute les instructions du bloc b.	

4.2.1 Syntaxe en Python

```
for variable in liste :
    instructions
```

Ici encore, la ligne contenant le mot-clé **for** doit se finir par un « **:** » et les instructions du bloc doivent être indentées. La fin de la boucle est marquée par un retour à la ligne non indenté.

4.2.2 Les intervalles d'entiers en Python

Pour répondre à la première question, il suffit de remarquer que les entiers de 0 à 19 par exemple, sont en fait les éléments d'une liste : `[0, 1, 2, ⋯ , 19]`. En Python, cette liste s'écrit de la manière suivante : `range(20)`.
Précisément, si a et b sont deux entiers, `range(a,b)` contient les éléments de l'intervalle semi-ouvert $[\![a, b[\![$, dans l'ordre croissant. Avec un seul argument, `range(b)` signifie `range(0,b)`.
Redisons-le, car c'est un fait important en Python : `range(a,b)` est intervalle **fermé** à gauche, **ouvert** à droite [1].

Avec `range`, nous pouvons maintenant itérer sur une liste d'entiers :

```
x = 5
for k in range(20):
    x = 2 * x - k - 3

print(x)
# Résultat obtenu : 1048600.
```

4.3 Boucles indéfinies ou conditionnelles

4.3.1 Algorithme

On peut aussi être amené à répéter un bloc d'instructions sans savoir combien de fois on devra le répéter.

Disposant d'une suite croissante, non majorée, on cherche à trouver le plus petit entier p tel que la valeur au rang p dépasse 10000.

Dans ce cas, on utilise la boucle **Tant que** qui permet de répéter le bloc d'instructions tant qu'une certaine condition est vérifiée.

1. Voir *Why numbering should start at zero*, E. W. Dijkstra, EWD831. Disponible en ligne.

	Tant que condition **faire** bloc d'instructions **Fin-du-Tant-que**
	signifie que
Tant que	la condition est vérifiée (expression booléenne=*True*)
Faire	le bloc d'instructions.

4.3.2 Syntaxe en Python

```
while condition :
    instructions
```

Rechercher le premier entier n tel que la somme des entiers de 1 à n dépasse 11.

```
n = 1
s = 1
while s < 11 :
    n = n + 1
    s = s + n

n
# REPONSE : n=5 (dans ce cas s=15)
```


5 Tableaux et type `list` en `python`

5.1 Deux structures de données en informatique.

Les données manipulées en informatique sont organisées, que ce soit dans la mémoire où dans la manière d'y accéder, de les manipuler. Une telle organisation porte le nom de **structure de données**, en voici deux grandes.

Tableau Les données sont stockées dans des cases contiguës de la mémoire de l'ordinateur, chaque emplacement étant souvent indicé par un entier. A priori, il n'est pas possible d'en rajouter autant que possible : cette place a été pré-allouée. Pour accéder au contenu de l'emplacement numéro k, la machine a seulement besoin de connaître l'adresse de la première case mémoire et de la largeur de chaque case (faire un dessin). L'accès en lecture et en écriture à une donnée (à partir du numéro de son emplacement) se fait donc en temps constant (on dit $O(1)$). L'ajout d'une nouvelle donnée à un tableau peut alors être problématique!

Liste (chaînée) Les données ne sont pas stockées de manière organisée dans la mémoire, mais de chaque emplacement on pointe l'emplacement suivant. Ainsi, l'accès (en lecture ou en écriture) ne se fait plus en temps constant, mais l'ajout d'une nouvelle donnée se fait simplement en temps constant.

5.2 Réalisation en `python`

Les objets de type `list` en Python sont des tableaux : c'est une dénomination fâcheuse car, partout ailleurs en informatique, le terme **liste** désigne en fait une liste chaînée. Prenez donc l'habitude de dire « tableau », ou « liste Python » et non « liste » quand vous parlez de cette structure.

Cependant, il y a une raison à cette dénomination : ces tableaux en Python possèdent presque la propriété des listes, dans le sens où l'ajout d'un nouvel élément se fait en temps constant **amorti**. Cela signifie que la plupart de ces ajouts se font en temps constants car en fait Python a « gardé des cellules en réserve », mais parfois l'ajout d'un élément force la création d'un nouveau tableau avec plus de cellules de réserve, et la copie de l'ancien tableau dans le nouveau. Cette opération est coûteuse, mais elle est rare. En moyenne, chaque ajout à un coût constant.

On parle alors de **tableaux dynamiques**.

Pour créer un tableau on peut le donner en extension.

```
t = [23, 41, 101]
t
```

On peut aussi le donner en compréhension.

```
u = [k**2 for k in range(10)]
u
```

On peut alors accéder à ses éléments via leur indice.

```
t[0]
u[1]
u[9]
```

Attention à utiliser un indice existant!

```
>>> t[3]
```

Les indices appartiennent à $[\![0, \texttt{len(t)}[\![$. Mais on peut aussi compter les éléments à partir de la fin, en utilisant des indices négatifs!

```
t[-1] # dernier élément
t[-2] # avant-dernier
```

```
t[-3]
```

Attention, là aussi, à ne pas sortir du tableau!

```
>>> t[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Enfin, il y a une liste Python particulière : la liste vide `[]`.

5.3 Opérations usuelles

5.3.1 Concaténation

Concaténer deux tableaux revient à les mettre bout à bout, on l'effectue en Python avec l'opérateur `+`.

```
t+u
u+t
```

Dans cette opération, les deux tableaux sont intégralement recopiés.

5.3.2 Quelques calculs

La longueur se calcule avec la fonction `len`. Pour des listes Python de nombres, le maximum se calcule avec la fonction `max`, le minimum avec `min` et la somme des éléments de la liste avec `sum`.

```
len(t)
max(t)
min(t)
sum(t)
```

On peut aussi tester l'appartenance d'un élément dans une liste Python avec `in`.

```
42 in t
[] in [[]]
```

Enfin, on peut recopier plusieurs fois un **même** tableau avec l'opérateur `*`.

```
4*t
t*4
```

5.3.3 Un peu de méthodes

Python est un langage **orienté objet** : en `python`, tout est un **objet**, et ces objets sont regroupées en **classes**. Une **méthode** est une fonction qui s'applique aux objets d'une classe particulière. Si `method` est une méthode de la classe `c`, et si `a` est un objet de cette classe, alors on applique `method` à `a` avec la syntaxe suivante : `a.method()`, les parenthèses pouvant contenir des paramètres.

On peut ajouter un élément à un tableau avec la méthode `append`. Le point important est que cette opération s'effectue en temps constant amorti.

```
t.append(-42)
t
```

À votre avis, pour un objet `x`, est-il équivalent d'effectuer `t+[x]` et `t.append(x)` ?

On peut aussi enlever le dernier élément d'une liste Python avec la méthode `pop()`.

```
x = t.pop()
t
x
```

Les autres méthodes disponibles sont rassemblées dans le tableau **??**. Elles ne sont pas exigibles.

Méthode	Description
`append(x)`	Ajoute `x` en fin de liste
`extend(L)`	Concatène la liste `L` en fin de liste
`insert(i,x)`	Insère `x` à la position `i`
`remove(x)`	Supprime la première occurence de `x` (erreur si impossible)
`pop(i)`	Supprime l'élément en position `i` (si vide, le dernier)
`index(x)`	Renvoie l'indice de la première occurence de `x` (erreur si impossible)
`count(x)`	Renvoie le nombre d'occurences de `x`
`sort(cmp,key,rev)`	Trie la liste (nombreuses options)
`reverse()`	Renverse la liste

TABLE 1 – Méthodes applicables aux listes.

5.4 Tranchage – Slicing

On peut directement accéder à une sous-liste Python (ou tranche —**slice** en anglais— c'est-à-dire bloc de cases consécutives) d'une liste, c'est ce que l'on appelle le tranchage (*slicing*).

On utilise les syntaxes

- `u[i:j]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`;
- `u[i:]` pour accéder à la tranche allant de l'indice `i` à la fin de la liste `u`;
- `u[:j]` pour accéder à la tranche allant du début de la liste `u` à l'indice `j-1`;
- `u[:]` pour accéder à la tranche complète (toute la liste `u`);
- `u[i:j:p]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`, par pas de `p`.

```
u
u[2:5]
u[2:]
u[:5]
u[1:8:2]
```

5.5 Tableaux multidimensionnels

On peut représenter une matrice avec des listes Python, par exemple en la décrivant ligne par ligne. Par exemple, on peut représenter la matrice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ par

```
M = [[1,2,3],[4,5,6],[7,8,9]]
M
M[0][2]
```

Cependant, cela ne permet pas d'effectuer les opérations classiques sur les matrices. On préférera utiliser les possibilités de la bibliothèque de calcul numérique `numpy`.

5.6 Alias

Parfois, lorsque l'on manipule des tableaux, on observe un phénomène étrange qui laisse pantois : c'est l'**aliasing**. C'est une notion assez subtile, mais qu'il est bon de connaître pour éviter les problèmes qui en découlent, ou au moins avoir une parade lorsqu'on s'y trouve confronté.

Commençons par un exemple où tout se passe intuitivement :

```
x = 3
y = x
x = 42
```

Que vaut alors `y` ?

```
y
```

Avec des listes Python, cela ne fonctionne pas exactement de cette façon :

```
x = [0]*5
y = x
x[0] = 42
```

Que vaut alors `y` ?

```
y
```

Tâchons de donner une explication à ce phénomène.

- Pour les objets de type **non mutable** (on ne peut pas les modifier, comme les types `int`, `float`, `bool`, `str`, `tuple`), une instruction du type `x = y` crée une nouvelle variable : la variable `x` pointe vers une case contenant la valeur de `y`, et l'on dit que `x` et `y` sont des **alias** de cette valeur. Réattribuer à `x` ou `y` une nouvelle valeur casse cet alias : les deux variables ne pointent plus vers la même valeur.
- Pour les objets de type **mutable**, les choses sont plus compliquées : on peut modifier leur contenu sans les réassigner. C'est le cas des listes `python`. Après une instruction du type `x = y` , les variables `x` et `y` pointent vers le même objet : ce sont des **alias** de cet objet là aussi. Mais si l'on modifie le contenu de `x`, sans réaffecter `x`, l'alias n'est pas cassé et l'on modifie à la fois `x` et `y`.

La fonction `id()`, qui affiche pour chaque objet son « numéro d'identité », permet de mettre cela en évidence :

```
x = 3
y = x
id(x),id(y)
x = 42
id(x),id(y)
```

```
x = [0]*5
y = x
id(x),id(y)
x[0] = 42
id(x),id(y)
```

Si l'on veut que `x` et `y` ne soient plus des alias, on pourra plutôt utiliser la méthode `copy()`.

```
x = [0]*5
y = x.copy()
id(x),id(y)
x[0] = 42
y
```

Il existe d'autres manières pour des tableaux de casser cet alias : `y = list(x)`, `y = x[:]` ou encore `y = x+[]`.

Cependant, en insérant des tableaux dans d'autres tableaux, on a une notion de « profondeur ». Les techniques données ci-dessus ne permettent de rompre un alias qu'au niveau de l'enveloppe externe. Il existe la fonction `deepcopy` de la bibliothèque `copy`, qui effectue une copie totale d'un objet, en profondeur comme son nom l'indique. Cela dit il est peu probable que nous en ayons besoin.

6 Chaînes et type `str` en `python`

6.1 Définition

Pour représenter des textes, on utilise la structure de données de « chaîne de caractères » (*string* en anglais). En `python`, cela correspond aux objets de type `str`.

Ces objets sont non-mutables, c'est-à-dire qu'une fois créés, on ne peut pas les modifier. Comme il ne peut y avoir des problèmes d'alias (par exemple, comme pour les listes `python`), nous ne détaillerons pas ici leur représentation en mémoire.

6.2 Création et lecture

On définit une chaîne de caractère en entourant ces caractères par des guillemets simples, doubles, ou trois guillemets simples ou doubles. L'utilisation de guillemets simples permet d'utiliser des guillemets doubles dans la chaîne de caractère, et vice-versa.

```
'Hallo Welt'
'It is "Hello World" !'
"Ja, aber 'not in germany' ..."
```

On peut signaler une tabulation par
`t` et créer une nouvelle ligne par
`n`.

```
s = u"\t Il faut une science politique nouvelle àun monde tout nouveau.\n[...]"
print(s)
```

```
    2
```

R La présence de `u` devant les guillements permet de s'assurer de prendre les comptes les caractères spéciaux liés à l'encodage utf-8.

On accède alors aux caractères d'une chaîne en donnant son indice, comme pour les listes.

```
s[0]
s[-1]
```

Remarquons que le tranchage fonctionne similairement aux listes.

```
s[14:21]
s[:11]
s[37:]
```

On peut aussi créer une chaîne de caractères à partir de nombres ou de booléens avec la fonction `str`.

```
str(42)
str(-3.5)
str(4==4.)
```

2. Alexis de Tocqueville, De la démocratie en Amérique (1835-1840)

Enfin, rappelons que ce type est non mutable.

```
>>> s[0] = "a"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s[0] = "a"
TypeError: 'str' object does not support item assignment
```

6.3 Opérations sur les chaînes

Comme pour les listes, on peut concaténer deux chaînes de caractères avec l'opérateur `+`.

```
'Hello'+'World'
```

On peut aussi dupliquer une chaîne de caractères plusieurs fois avec l'opérateur `*`.

```
print('GA-BU-ZO-MEU\n'*5)
```

La fonction `len` permet là encore de calculer la longueur d'une liste.

```
len(s)
```

On peut aussi chercher la présence d'un motif dans une chaîne avec `in`, son absence avec `not in`

```
'Science' in s
'science' not in s
```

6.4 Chaînes et méthodes

De nombreuses méthodes existent sur les chaînes, voici les plus utiles.

Méthode	Description
`count(m)`	Compte le nombre d'occurrences du motif `m`, sans chevauchement.
`find(m)`	Renvoie la première occurrence du motif `m`.
`islower() (et autres)`	Renvoie un booléen indiquant si la chaîne est en minuscules.
`join(bout)`	Concatène les éléments de la liste `bout`, séparés par la chaîne.
`lower()`	Met en minuscule la chaîne de caractères.
`split(sep)`	Sépare la chaîne selon le séparateur `sep`
`strip(char)`	Enlève les lettres en début/fin si elles sont dans la chaîne `char`.
`upper()`	Mets en majuscules la chaîne.

```
s = " que ; d'espaces ; mes ; amis ! ; "
s.count('es')
s.find('es')
s.islower()
'+'.join([str(i) for i in range(14)])
'HAHAHAHAHA !'.lower()
s.split(';')
s.split()
s.strip()
s.strip('q; ')
s.upper()
```