Python pour CPGE scientifiques Documentation

Version 1

Laurent Garcin

mars 29, 2018

Contenu:

1 Entrée/sortie **1**
- 1.1 Interaction directe avec l'utilisateur . . . 1
- 1.2 Fichiers . . . 2

2 Types, opérateurs et variables **5**
- 2.1 Variables . . . 5
- 2.2 Types . . . 7
- 2.3 Opérateurs . . . 8
- 2.4 Conversions de types . . . 10

3 Types composés **11**
- 3.1 Listes . . . 11
- 3.2 Tuples . . . 15
- 3.3 Chaînes de caractère . . . 18
- 3.4 Mutabilité . . . 19
- 3.5 Dictionnaires . . . 24

4 Rudiments d'algorithmique **25**
- 4.1 Tests . . . 25
- 4.2 Boucles . . . 27
- 4.3 Fonctions . . . 30

5 Algorithmes classiques **41**
- 5.1 Algorithmes de recherche . . . 41
- 5.2 Analyse numérique . . . 46
- 5.3 Arithmétique . . . 54
- 5.4 Probabilités . . . 58
- 5.5 Matrices . . . 61
- 5.6 Tris . . . 65

6 Un peu de théorie **69**
- 6.1 Preuve d'un algorithme . . . 69
- 6.2 Complexité . . . 70

7 Index et tables **71**

CHAPITRE 1

Entrée/sortie

Un programme informatique doit souvent pouvoir interagir avec l'extérieur. Il existe plusieurs manières de faire cela :

— soit en interagissant directement avec l'utilisateur : celui-ci peut rentrer des données grâce au clavier ou voir le résultat du programme s'afficher à l'écran ;
— soit au moyen de fichiers : le programme peut lire des informations dans des fichiers ou au contraire écrire des résultats dans des fichiers.

1.1 Interaction directe avec l'utilisateur

L'interpéteur **IPython** affiche par défaut le résultat d'une expression non évaluée à `None`. Ce n'est pas le cas lorsque l'on exécute un script Python. Si l'on veut afficher des résultats à l'écran, il faut explicitement utiliser la fonction `print`.

Dans cet exemple, on exécute le script stocké dans le fichier `script_print.py`.

```
'titi'          # Cette ligne n'affiche rien
print('toto')
```

```
$ python script_print.py
toto
```

Inversement lorsque l'on souhaite demander une information à l'utilisateur, on utilise la fonction `input`. Celle-ci interrompt l'exécution du programme, ce qui permet à l'utilsateur d'entrer des caractères au clavier. Pour terminer sa saisie, l'utilisateur appuie alors sur la touche `Entrée`. La fonction `input` renvoie alors la chaîne de caractères entrée par l'utilisateur et l'exécution du programme reprend son cours. On peut également fournir en argument optionnel à la fonction `input` un message à destination de l'utilisateur qui sera affiché avant que celui-ci commence à saisir des caractères au clavier.

```
In [1]: result = input('Quel est votre nom ?')
Quel est votre nom ?
# Ici l'utilisateur a entré Toto
In [2]: result
Out[2]: 'Toto'
```

1.2 Fichiers

Pour ouvrir un fichier, on utilise la fonction `open` en donnant en argument le chemin du fichier. On peut préciser plusieurs paramètres optionnels.

`mode` Mode d'ouverture du fichier : essentiellement `'r'` pour lecture, `'w'` pour écriture et `'r+'` pour lecture/écriture. Par défaut, un fichier est ouvert en lecture seulement.

`encoding` Encodage[1] du fichier : de nombreux encodages sont disponibles comme `'utf8'` (à privilégier) ou `'ascii'`.

```
In [3]: f = open('test.txt', mode='r', encoding='utf8')

In [4]: f
Out[4]: <_io.TextIOWrapper name='test.txt' mode='r' encoding='utf8'>

In [5]: type(f)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]: _io.
↪TextIOWrapper
```

Avertissement : Le mode d'écriture `'w'` crée un nouveau fichier si un fichier du même nom n'existe pas mais « écrase » un fichier existant dans le cas contraire.

L'objet renvoyé par open dispose d'une méthode `close` pour fermer le fichier. On ne pourra alors plus lire ou écrire dedans.

```
In [6]: f.close()
```

L'objet renvoyé par `open` dispose de plusieurs méthodes permettant de lire ou d'écrire dans le fichier ouvert.

On utilisera pour les exemples le fichier `test.txt` suivant.

```
Première ligne
Deuxième ligne
Troisième ligne
```

Dans les exemples suivants, le caractère `\n` désigne un *retour à la ligne*.

```
In [7]: f = open('test.txt', mode='r', encoding='utf8')

# Renvoie l'intégralité d'un fichier
In [8]: f.read()
Out[8]: 'Première ligne\nDeuxième ligne\nTroisième ligne\n'

In [9]: f.close()

In [10]: f = open('test.txt', mode='r', encoding='utf8')

# Renvoie la liste des lignes
```

[1] Un fichier n'est qu'une suite de bits stocké en mémoire. L'encodage permet de transformer cette suite de bits en une suite de caractères et inversement. C'est en quelque sorte une table qui à une suite de bits associe un caractère. Il faut connaître l'encodage d'un fichier texte pour pouvoir le lire correctement. Certains encodages ne permettent pas de représenter certains caractères. C'est pour cela qu'on préconise l'utilisation de l'encodage UTF-8 qui permet de représenter tous les caractères relatifs à chacune des langues de la planète.

```
In [11]: f.readlines()
Out[11]: ['Première ligne\n', 'Deuxième ligne\n', 'Troisième ligne\n']

In [12]: f.close()

In [13]: f = open('toto.txt', mode='w', encoding='utf8')

In [14]: f.write("Ce qui se conçoit bien s'énonce clairement\n")
Out[14]: 43

In [15]: f.write("Et les mots pour le dire viennent aisément.\n")
\\\\\\\\\\\\Out[15]: 44

In [16]: f.close()
```

On vérifie que le fichier `toto.txt` a bien été créé et que son contenu correspond bien à ce qu'on y a écrit.

```
Ce qui se conçoit bien s'énonce clairement
Et les mots pour le dire viennent aisément.
```

CHAPITRE 2

Types, opérateurs et variables

2.1 Variables

2.1.1 Variable et affectation

Une *variable* est un objet qui associe un nom (*identifiant*) à un objet stocké en mémoire. Concrètement, une variable permet de conserver un objet en vue d'une utilisation ultérieure.

Pour stocker un objet dans une variable, on utilise l'opérateur d'affectation =.

```
In [1]: a = 2

In [2]: b = 3

In [3]: (a + b)**2
Out[3]: 25
```

Avertissement : Le symbole = n'a pas du tout le même sens qu'en mathématiques. En mathématiques, ce symbole désigne un état de fait et a une valeur logique (une égalité peut être vraie ou fausse) tandis qu'en Python, il accomplit une action (stocker une valeur dans une variable).

Bien entendu, on peut changer la valeur d'une variable (d'où le nom) et même le type de cette valeur.

```
In [4]: a = 42

In [5]: a
Out[5]: 42

In [6]: a = "toto"

In [7]: a
Out[7]: 'toto'
```

Echange de deux variables

On est souvent amené à échanger les valeurs de deux variables. La méthode naïve ne fonctionne pas.

```
In [8]: a = 1

In [9]: b = 2

In [10]: a = b # Les deux variables valent 2 !

In [11]: b = a

In [12]: a
Out[12]: 2

In [13]: b
\\\\\\\\\\Out[13]: 2
```

L'idée est alors d'employer une variable auxiliaire.

```
In [14]: a = 1

In [15]: b = 2

In [16]: aux = a

In [17]: a = b

In [18]: b = aux

In [19]: a
Out[19]: 2

In [20]: b
\\\\\\\\\\Out[20]: 1
```

On peut également utiliser des *affectations multiples* à l'aide de tuples ou de listes.

2.1.2 Identifiant d'une variable

Bien que cela n'ait rien d'obligatoire, il est judicieux que l'identifiant d'une variable donne une indication quant à l'utilisation de cette variable. Par exemple, une variable devant contenir la *moyenne* de plusieurs nombres sera plus judicieusement nommée `moyenne` ou même `moy` plutôt que `x` ou `toto`.

Note : On ne peut pas choisir n'importe quel identifiant pour une variable.

— Certains mots du langage Python sont dits « réservés » et ne peuvent servir à désigner une variable (`for`, `while`, `del`, ...).

— Le nommage des identifiants doit obéir à certaines règles : le premier caractère ne peut être un chiffre, certains caractères comme `#` ou `@` sont proscrits, ...

2.2 Types

On décrit dans ce paragraphe les types de base utilisés par Python.

2.2.1 Booléen

Un objet de type *booléen* peut prendre les valeurs logiques **vrai** (`True`) ou **faux** (`False`).

```
In [1]: type(True)
Out[1]: bool

In [2]: type(False)
\\\\\\\\\\\\\Out[2]: bool
```

2.2.2 Entier

Les entiers Python ne sont théoriquement pas limités en taille même si, en pratique, on est limité par la mémoire de la machine.

```
In [3]: type(12)
Out[3]: int

In [4]: type(-34)
\\\\\\\\\\\\\Out[4]: int
```

2.2.3 Flottant

Les nombres non entiers (nombres « à virgule ») sont stockés sous la forme de nombres *flottants*. La virgule est notée à l'anglo-saxonne à l'aide du caractère `.`. On peut également utiliser la notation scientifique : `1.23e45` signifie $1,23 \times 10^{45}$.

```
In [5]: type(1.23)
Out[5]: float

In [6]: type(1.)
\\\\\\\\\\\\\\Out[6]: float

In [7]: type(1.23e45)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: float
```

Les nombres flottants sont en fait de la forme $\pm m.2^e$ où m est un réel de l'intervalle $[1, 2[$ appelé mantisse et e est un entier. Les nombres m et e sont stockés sur un nombre fixé de bits [1] : il n'existe par conséquent qu'un nombre **fini** de nombres flottants.

Il faut bien comprendre que les nombres flottants ne sont en fait que des **approximations** des nombres réels que l'on considère, ce qui peut parfois créer des surprises.

```
In [8]: 2.3-1.5
Out[8]: 0.7999999999999998
```

[1] Pour plus de précision, on pourra consulter la définition de la norme IEEE 754.

2.2.4 Complexe

Les nombres complexes se notent à l'aide de la lettre `j` qui indique la partie imaginaire. Par exemple, le complexe $1 + 2i$ s'écrit `1+2j`.

```
In [9]: type(1.2+3.4j)
Out[9]: complex
```

Note : Pour représenter un complexe la lettre `j` doit toujours être précédée d'un caractère numérique. Le complexe i sera donc représenté par `1j`.

On peut accéder aux parties réelle et imaginaire d'un nombre complexe grâce aux attributs `real` et `imag`.

```
In [10]: z = 3 + 4j

In [11]: z.real
Out[11]: 3.0

In [12]: z.imag
\\\\\\\\\\\\Out[12]: 4.0
```

2.2.5 Dictionnaire (hors programme)

```
In [13]: type({'toto': 123, 'titi': 4.56, 'tata': 'abc'})
Out[13]: dict
```

Notes

2.3 Opérateurs

2.3.1 Opérateurs arithmétiques

Python dispose des opérateurs arithmétiques de base + (addition), − (soustraction), $\star$ (multiplication) et / (division).

```
In [1]: 1 + 2.3 * (4.5 - 6) / 7.8
Out[1]: 0.5576923076923077
```

L'exponentiation se fait à l'aide de l'opérateur $\star\star$.

```
In [2]: 8**3
Out[2]: 512
```

Par ailleurs, les opérateurs `%` et `//` calculent respectivement le quotient et le reste d'une division euclidienne.

```
In [3]: 20%6
Out[3]: 2

In [4]: 20//6
\\\\\\\\\Out[4]: 3
```

2.3.2 Opérateurs de comparaison

Le résultat d'une comparaison est un booléen. Python dispose des opérateurs de comparaison <, >, <=, >=, == (égalité), != (différence).

```
In [5]: 1 + 2 < 3.4
Out[5]: True

In [6]: 5.6 >= 7
\\\\\\\\\\\\\Out[6]: False

In [7]: 1 + 1 == 2
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: True

In [8]: "abc" != "def"
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]: True
```

Note : Il faut prendre garde de ne pas confondre l'opérateur d'**égalité** == avec l'opérateur d'**affectation** =.

2.3.3 Opérateurs logiques

On dispose des opérateurs logique classiques : `not` (négation), `and` (conjonction) et `or` (disjonction).

```
In [9]: not False
Out[9]: True

In [10]: 1+2<3 and 4.5==6
\\\\\\\\\\\\\Out[10]: False

In [11]: 7<=8 or "abc"=="def"
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]: True
```

2.3.4 Opération/affectation

Les opérateurs binaires peuvent être suivis du symbole = pour effectuer simultanément une opération et une affectation. Ainsi `a += b` équivaut à `a = a + b`

```
In [12]: a = 2

In [13]: a += 3

In [14]: a
Out[14]: 5

In [15]: a *= 2

In [16]: a
Out[16]: 10

In [17]: a **= 3

In [18]: a
Out[18]: 1000
```

```
In [19]: a %= 30

In [20]: a
Out[20]: 10
```

2.4 Conversions de types

On peut convertir un objet d'un type vers un autre type : il suffit pour cela d'utiliser le nom du type vers lequel on veut convertir. On parle alors de *conversion explicite*.

```
In [1]: int('42')
Out[1]: 42

In [2]: type(int('42'))
\\\\\\\\\\Out[2]: int

In [3]: str(25)
\\\\\\\\\\\\\\\\\\\\\\Out[3]: '25'

In [4]: type(str(25))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]: str
```

Il se peut que certaines conversions soient impossibles.

```
In [5]: int('45f6')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-b58d830d7e7d> in <module>()
----> 1 int('45f6')

ValueError: invalid literal for int() with base 10: '45f6'
```

L'utilisation d'opérateurs pour des objets dont le type n'est pas approprié peut donner lieu à des *conversions implicites*.

```
# Le booléen True est automatiquement converti en l'entier 1 pour pouvoir l'ajouter à␣
↪l'entier 2
In [6]: True + 2
Out[6]: 3
```

Mais il se peut qu'aucune conversion implicite ne puisse avoir lieu, ce qui conduit alors à une erreur.

```
In [7]: 'a' + 1.23
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-56a632500eb2> in <module>()
----> 1 'a' + 1.23

TypeError: Can't convert 'float' object to str implicitly
```

CHAPITRE 3

Types composés

3.1 Listes

3.1.1 Création de listes et opérations de base

Une **liste** est tout simplement une collection d'objets. On la déclare en séparant ses éléments par des virgules `,` et en les encadrant par des crochets `[...]`.

```
In [1]: type([1,2,3])
Out[1]: list
```

Les objets contenus dans une même liste peuvent être de types différents.

```
In [2]: [1.23, 'abc', 45]
Out[2]: [1.23, 'abc', 45]
```

On peut même imbriquer des listes.

```
In [3]: [1.23, ["abc", "def", "ghi"], [45, 67]]
Out[3]: [1.23, ['abc', 'def', 'ghi'], [45, 67]]
```

L'opérateur + permet de *concaténer* des listes.

```
In [4]: [1.23, 'abc', 45] + [6, 'def', 'ghi', 7.89]
Out[4]: [1.23, 'abc', 45, 6, 'def', 'ghi', 7.89]
```

On devine alors l'action de l'opérateur $\star$ [1].

```
In [5]: [1.23, 'abc', 45] * 3
Out[5]: [1.23, 'abc', 45, 1.23, 'abc', 45, 1.23, 'abc', 45]
```

[1] En termes savants, l'ensemble des listes munis de la loi + est un *monoïde*. La loi + est en effet une loi interne associative et la liste vide `[]` est neutre pour cette loi. Le « produit » d'une liste par un entier (positif) n'est autre qu'un *multiple* de cette liste.

Note : On ne peut évidemment multiplier une liste que par un **entier**.

La fonction `len` permet de récupérer la longueur d'une liste.

```
In [6]: len([1.23, 'abc', 45])
Out[6]: 3
```

3.1.2 Accès aux éléments

On peut accéder aux éléments d'une liste via leurs indices et l'opérateur `[ ]`.

```
In [7]: ma_liste = [25, 34, 48, 67]

In [8]: ma_liste[2]
Out[8]: 48
```

Avertissement : Il est à noter que les indices des listes commencent à 0 et non 1. Le dernier indice d'une liste de n éléments est donc $n-1$ et non n.

On peut bien sûr accéder à des éléments de listes imbriquées.

```
In [9]: ma_liste = [1.23, ["abc", "def", "ghi"], [45, 67]]

In [10]: ma_liste[1][2]
Out[10]: 'ghi'
```

On peut également accéder à des éléments d'une liste « par la fin ».

```
In [11]: ma_liste = ['a', 'b', 'c', 'd', 'e']

In [12]: ma_liste[-1]
Out[12]: 'e'

In [13]: ma_liste[-3]
\\\\\\\\\\\\Out[13]: 'c'
```

3.1.3 Modification des éléments

L'opérateur `[ ]` permet également de modifier les éléments d'une liste.

```
In [14]: ma_liste = [25, 34, 48, 67]

In [15]: ma_liste[2] = 666

In [16]: ma_liste
Out[16]: [25, 34, 666, 67]
```

Bien évidemment, cela fonctionne aussi pour les listes imbriquées.

```
In [17]: ma_liste = [1.23, ["abc", "def", "ghi"], [45, 67]]

In [18]: ma_liste[1][2] = "toto"

In [19]: ma_liste
Out[19]: [1.23, ['abc', 'def', 'toto'], [45, 67]]
```

3.1.4 Insertion et suppression d'éléments

Il existe plusieurs moyens d'ajouter des éléments à une liste.

La première méthode est de les ajouter un par un grâce aux méthodes `append` (insertion en fin de liste) ou `insert` (insertion à un endroit donné).

```
In [20]: ma_liste = ['a', 1, 'b']

In [21]: ma_liste.append(2)

In [22]: ma_liste
Out[22]: ['a', 1, 'b', 2]

In [23]: ma_liste.insert(2, 'toto')

In [24]: ma_liste
Out[24]: ['a', 1, 'toto', 'b', 2]
```

Pour ajouter plusieurs éléments d'affilée, on peut utiliser l'opérateur de concaténation + ou de concaténation/affectation += ou encore la méthode `append`.

```
In [25]: ma_liste = ['a', 1, 'b', 2]

In [26]: ma_liste = ma_liste + ['c', 3, 'd']

In [27]: ma_liste
Out[27]: ['a', 1, 'b', 2, 'c', 3, 'd']

In [28]: ma_liste += [4, 5]

In [29]: ma_liste
Out[29]: ['a', 1, 'b', 2, 'c', 3, 'd', 4, 5]

In [30]: ma_liste.extend(['e', 6, 'f'])

In [31]: ma_liste
Out[31]: ['a', 1, 'b', 2, 'c', 3, 'd', 4, 5, 'e', 6, 'f']
```

De même, il existe plusieurs façons de supprimer des éléments d'une liste.

Pour supprimer des éléments, on peut utiliser les méthodes `pop` (renvoie le dernier élément et le supprime de la liste) ou `remove` (supprime un élément de valeur donnée).

```
In [32]: ma_liste = ['a', 1, 'b', 2, 'c', 3]

In [33]: ma_liste.pop()
Out[33]: 3

In [34]: ma_liste
```

```
\\\\\\\\\\Out[34]: ['a', 1, 'b', 2, 'c']

In [35]: ma_liste.remove('b')

In [36]: ma_liste
Out[36]: ['a', 1, 2, 'c']
```

Note : La méthode `remove` ne supprime que la *première* occurence d'une valeur donnée.

```
In [37]: ma_liste = [1, 2, 3, 2, 4, 2]

In [38]: ma_liste.remove(2)

In [39]: ma_liste
Out[39]: [1, 3, 2, 4, 2]
```

La suppression d'éléments peut également se faire au moyen du mot-clé `del` [2].

```
In [40]: ma_liste = ['a', 1, 'b', 2, 'c', 3]

In [41]: del ma_liste[2]

In [42]: ma_liste
Out[42]: ['a', 1, 2, 'c', 3]
```

3.1.5 Sous-listes (slicing)

Il existe une syntaxe permettant de créer une sous-liste d'une liste.

```
In [43]: ma_liste = ['a', 'b', 'c', 'd', 'e', 'f']

In [44]: ma_liste[2:5]
Out[44]: ['c', 'd', 'e']
```

De manière générale, si `li` est une liste, alors `li[a:b]` renvoie la liste formée des éléments de la liste `li` dont les indices sont compris entre `a` (**inclus**) et `b` (**non inclus**).

Si `a` ou `b` sont omis, la sélection s'opére à partir du début ou jusqu'à la fin de la liste.

```
In [45]: ma_liste = ['a', 'b', 'c', 'd', 'e', 'f']

In [46]: ma_liste[2:]
```

De manière générale, le mot-clé `del` « supprime » une variable (sans rentrer dans les détails).

```
In [59]: a = 42

In [60]: del a

In [61]: a
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-61-3f786850e387> in <module>()
----> 1 a

NameError: name 'a' is not defined
```

```
Out[46]: ['c', 'd', 'e', 'f']

In [47]: ma_liste[:4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]: ['a', 'b', 'c', 'd']
```

On peut encore sélectionner des éléments à intervalles réguliers.

```
In [48]: ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']

In [49]: ma_liste[2:9:3]
Out[49]: ['c', 'f', 'i']

In [50]: ma_liste[7:2:-2]
\\\\\\\\\\\\\\\\\\\\\\\\Out[50]: ['h', 'f', 'd']
```

Le slicing permet aussi de modifier les éléments d'une liste.

```
In [51]: ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']

In [52]: ma_liste[2:9:3]
Out[52]: ['c', 'f', 'i']

In [53]: ma_liste[2:9:3] = 'toto', 'tata', 'titi'

In [54]: ma_liste
Out[54]: ['a', 'b', 'toto', 'd', 'e', 'tata', 'g', 'h', 'titi', 'j', 'k', 'l']
```

On peut combiner le slicing et le mot-clé `del` pour supprimer plusieurs éléments à la fois.

```
In [55]: ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']

In [56]: ma_liste[5:10:2]
Out[56]: ['f', 'h', 'j']

In [57]: del ma_liste[5:10:2]

In [58]: ma_liste
Out[58]: ['a', 'b', 'c', 'd', 'e', 'g', 'i', 'k', 'l']
```

Notes

3.2 Tuples

Un *tuple* est tout à fait similaire à une liste : il s'agit d'une collection d'objets [1] . On crée un tuple en séparant les objets par des des virgules `,` et en les encadrant *éventuellement* par des parenthèses `(...)`.

```
In [1]: type((1,2,3))
Out[1]: tuple

In [2]: a = 1,2,3
```

[1] Il s'agit exactement de la notion de *uplet* en mathématiques. L'appellation *tuple* provient en fait de l'anglais. En effet, en anglais, on parle de « quadruple », « quintuple », etc.. et plus généralement de n-**tuple** tandis qu'en français, on emploie les termes « quadruplet », « quintuplet », etc.. et de manière générale n-**uplet**. Néanmoins, la terminologie anglo-saxonne s'est imposée en ce qui concerne Python.

```
In [3]: type(a)
Out[3]: tuple
```

Quand faut-il encadrer un tuple avec des parenthèses ? Par exemple si l'on crée une liste dont certains arguments sont des tuples.

```
In [4]: [(1, 2), (3, 4)]
Out[4]: [(1, 2), (3, 4)]

In [5]: [1, 2, 3, 4]
\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]: [1, 2, 3, 4]

In [6]: [(1, 2), (3, 4)] == [1, 2, 3, 4]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]: False
```

Ou bien quand on on apelle une fonction dont un ou plusieurs arguments sont des tuples.

```
f((1, 2), (3, 4))    # fonction de deux arguments (deux tuples)
f(1, 2, 3, 4)        # fonction de quatre arguments (quatre entiers)
```

Les opérateurs + et += fonctionnent comme pour les listes.

```
In [7]: (1,2) + (3,4)
Out[7]: (1, 2, 3, 4)

In [8]: a = (1,2)

In [9]: a += (3,4)

In [10]: a
Out[10]: (1, 2, 3, 4)
```

On peut **accéder** aux éléments d'un tuple de la même manière qu'on accède aux éléments d'une liste.

```
In [11]: a = (1, 'a', 2, 'b', 3, 'c', 4, 'd', 5, 'e', 6, 'f')

In [12]: a[3]
Out[12]: 'b'

In [13]: a[2:]
\\\\\\\\\\\\\Out[13]: (2, 'b', 3, 'c', 4, 'd', 5, 'e', 6, 'f')

In [14]: a[:5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: (1, 'a', 2, 'b
↪', 3)

In [15]: a[-3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]:
↪'e'

In [16]: a[2:9:2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪(2, 3, 4, 5)

In [17]: a[8:1:-3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪(5, 'c', 2)
```

Par contre, on ne peut pas **modifier** un tuple : on ne peut ni modifier ses éléments, ni en ajouter, ni en enlever.

```
s = (1, 2, 3)
s[0] = 4
```

```
In [18]: s = (1, 2, 3)

In [19]: s[0] = 4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-06ee6658ee71> in <module>()
----> 1 s[0] = 4

TypeError: 'tuple' object does not support item assignment
```

A nouveau, la fonction `len` renvoie la longueur d'un tuple.

```
In [20]: len((1.23, 'abc', 45))
Out[20]: 3
```

Affectations multiples

Les tuples permettent d'affecter des valeurs à plusieurs variables en même temps [2].

```
In [21]: a, b, c = 1, 2, 3

In [22]: a
Out[22]: 1

In [23]: b
\\\\\\\\\\Out[23]: 2

In [24]: c
\\\\\\\\\\\\\\\\\\\\Out[24]: 3
```

Cela permet notamment d'échanger élégamment les valeurs de deux variables [3].

```
In [25]: a, b = 1, 2

In [26]: a
Out[26]: 1

In [27]: b
\\\\\\\\\\Out[27]: 2

In [28]: a, b = b, a

In [29]: a
Out[29]: 2

In [30]: b
\\\\\\\\\\Out[30]: 1
```

On peut également procéder à des affectations multiples à l'aide de listes.

Notes

3.3 Chaînes de caractère

Une chaîne de caractères n'est autre qu'une liste de caractères. Les caractères sont juxtaposés et encadrés par des guillemets simples `'...'` ou doubles `"..."`.

```
In [1]: type('abcdef')
Out[1]: str

In [2]: "abcdef" == 'abcdef'
\\\\\\\\\\\\Out[2]: True
```

A nouveau, les opérateurs + et += fonctionnent comme pour les listes.

```
In [3]: 'abc' + "def"
Out[3]: 'abcdef'

In [4]: s = 'abc'

In [5]: s += "def"
```

On peut **accéder** de la même manière aux caractères d'une chaîne qu'aux éléments d'une liste.

```
In [6]: ma_chaine = "Bonjour Python"

In [7]: ma_chaine[3]
Out[7]: 'j'

In [8]: ma_chaine[2:]
\\\\\\\\\\\\Out[8]: 'njour Python'
```

```
In [31]: [a, b, c] = [1, 2, 3]

In [32]: a
Out[32]: 1

In [33]: b
\\\\\\\\\\\Out[33]: 2

In [34]: c
\\\\\\\\\\\\\\\\\\\\\\\Out[34]: 3
```

A nouveau, on peut également utiliser des listes pour échanger les valeurs de deux variables.

```
In [35]: [a, b] = [1, 2]

In [36]: a
Out[36]: 1

In [37]: b
\\\\\\\\\\\Out[37]: 2

In [38]: [a, b] = [b, a]

In [39]: a
Out[39]: 2

In [40]: b
\\\\\\\\\\\Out[40]: 1
```

```
In [9]: ma_chaine[:5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]: 'Bonjo'

In [10]: ma_chaine[-3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]: 'h'

In [11]: ma_chaine[2:9:2]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]: 'norP'

In [12]: ma_chaine[8:1:-3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:␣
↪'Pun'
```

Par contre, on ne peut pas **modifier** une chaîne.

```
In [13]: s = 'abc'

In [14]: s[0] = 'z'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-5994e1a74598> in <module>()
----> 1 s[0] = 'z'

TypeError: 'str' object does not support item assignment
```

Encore une fois, la fonction `len` renvoie la longueur d'une chaîne de caractères.

```
In [15]: len('Anticonstitutionnellement')
Out[15]: 25
```

Les chaînes possèdent de nombreuses méthodes : ces méthodes ne modifient jamais la chaîne de caractère mais renvoient une *autre* chaîne de caractère.

```
In [16]: ma_chaine = 'Mon nom est Bond. James Bond.'

In [17]: ma_chaine.replace('Bond', 'Toto')
Out[17]: 'Mon nom est Toto. James Toto.'

In [18]: ma_chaine
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]: 'Mon nom est Bond. James Bond.'

In [19]: ma_chaine.upper()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:␣
↪'MON NOM EST BOND. JAMES BOND.'

In [20]: ma_chaine
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'Mon nom est Bond. James Bond.'
```

3.4 Mutabilité

Il s'agit d'un paragraphe un peu subtil : il s'agit d'expliquer la différence fondamentale qu'il existe en Python entre les objets que l'on peut modifier (listes) ou que l'on ne peut modifier (tuples ou chaînes de caractère).

Considérons ce premier exemple où les variables sont des entiers.

```
In [1]: a = 1

In [2]: b = a

In [3]: a = 2            # on modifie a

In [4]: a
Out[4]: 2

In [5]: b                # b n'a pas ete modifiée
\\\\\\\\\\Out[5]: 1
```

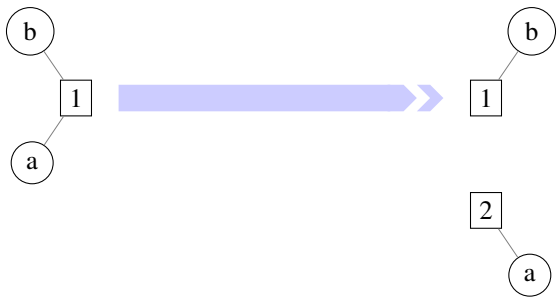
Considérons maintenant l'exemple suivant où les variables sont des listes.

```
In [6]: a = [1, 2, 3]

In [7]: b = a

In [8]: a[0] = 'foo'     # on modifie la liste a

In [9]: a
Out[9]: ['foo', 2, 3]

In [10]: b                # la liste b a aussi ete modifiée !
\\\\\\\\\\\\\\\\\\\\\\Out[10]: ['foo', 2, 3]
```

Pour expliquer la différence entre ces deux exemples, il faut comprendre la représentation des objets Python en mémoire. Pour cela, on va utiliser la fonction `id`. Pour schématiser, celle-ci renvoie l'emplacement en mémoire d'un objet.

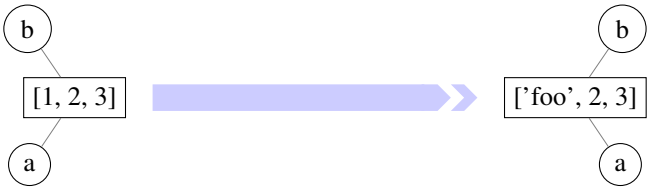
```
In [11]: a = 1

In [12]: b = a

In [13]: id(a), id(b)     # les variables a et b pointent vers le même emplacement en␣
↪mémoire
Out[13]: (10911168, 10911168)

In [14]: a = 2

In [15]: id(a), id(b)     # la variable b pointe toujours vers le même emplacement␣
↪mais plus la variable b
Out[15]: (10911200, 10911168)
```

L'instruction `a = 2` a fait pointer la variable `a` vers un autre emplacement en mémoire où est stocké l'entier `2`.



```
In [16]: a = [1, 2, 3]

In [17]: b = a

In [18]: id(a), id(b)     # les variables a et b pointent vers le même emplacement en
↪mémoire
Out[18]: (140367336740104, 140367336740104)

In [19]: a[0] = 'foo'

In [20]: id(a), id(b)     # les variables a et b pointent toujours vers le même
↪emplacement
Out[20]: (140367336740104, 140367336740104)
```

Ici, l'instruction `a[0] = 'foo'` a modifié l'objet stocké à l'emplacement commun vers lequel pointent les variables `a` et `b`. Comme `a` et `b` pointent toujours le même emplacement en mémoire, la variable `b` est maintenant associée à ce nouvel objet.



Mais pourquoi cette différence de comportement ? Il existe en Python deux types d'objets : les objets **mutables** et les objets **immutables**. On peut donner la définition suivante.

> Un objet est dit **mutable** si on peut changer sa valeur après sa création. Il est dit **immutable** dans le cas contraire [1].

Objets immutables Entiers, flottants, complexes, tuples, chaînes de caractères, ...

Objets mutables Listes, dictionnaires, ...

Voilà la solution du mystère : toutes les variables pointant vers un même objet mutable sont affectées par la modification de cet objet. Ceci ne peut pas se produire lorsque des variables pointent vers un objet immutable puisque celui-ci ne peut-être modifié.

Note : Bien souvent, on veut copier une liste dans un nouvel objet pour qu'il ne subisse pas les modifications de l'objet initial. Pour cela, il ya plusieurs possibilités :

[1] Ce n'est pas rigoureusement exact. Un objet immutable tel qu'un tuple peut contenir des objets mutables comme des listes. Néanmoins, chaque objet du tuple conserve le même emplacement en mémoire même s'il a été modifié.

```
In [61]: a = ([1, 2, 3], 'toto', 'tata')

In [62]: b = a

In [63]: a[0][1] = 1000

In [64]: a
Out[64]: ([1, 1000, 3], 'toto', 'tata')

In [65]: b                    # b a egalement ete modifié
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[65]: ([1, 1000, 3], 'toto', 'tata')

In [66]: id(a[0]), id(b[0])  # le premier élément du tuple est toujours le même
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[66]:
↪(140367336372168, 140367336372168)
```

— le slicing `[:]` ;
— l'utilisation de la méthode `copy` ;
— l'utilisation du constructeur `list`.

```
In [21]: liste1 = [1, 2, 3]

In [22]: liste2 = liste1[:]

In [23]: liste3 = liste1.copy()

In [24]: liste4 = list(liste1)

In [25]: id(liste1), id(liste2), id(liste3), id(liste4)  # les objets sont bien␣
↪distincts
Out[25]: (140367337003528, 140367337003912, 140367336853256, 140367337086856)

In [26]: liste1[0] = 'toto'

In [27]: liste1, liste2, liste3, liste4                  # liste1 a ete modifiée mais␣
↪pas les autres listes
Out[27]: (['toto', 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3])
```

Les opérateurs + et +=

Le lecteur attentif aura remarqué qu'on semblerait pouvoir modifier un objet immutable telle qu'une chaîne de caractères ou une liste à l'aide des opérateurs + ou +=. Mais ces opérateurs ne modifient pas l'objet en question ; ils créent en fait un **nouvel** objet. On peut s'en convaincre à l'aide de la fonction `id`.

```
In [28]: t = (1, 2, 3)

In [29]: id(t)
Out[29]: 140367337345816

In [30]: t = t + (4, 5)

In [31]: t
Out[31]: (1, 2, 3, 4, 5)

In [32]: id(t)
\\\\\\\\\\\\\\\\\\\\\\\\Out[32]: 140367480094656
```

```
In [33]: t = (1, 2, 3)

In [34]: id(t)
Out[34]: 140367337335400

In [35]: t += (4, 5)

In [36]: t
Out[36]: (1, 2, 3, 4, 5)

In [37]: id(t)
\\\\\\\\\\\\\\\\\\\\\\\\Out[37]: 140367424121232
```

Pour les objets mutables tels que les listes, les opérateurs + et += se comportent de manières différentes : l'opérateur

+ crée un nouvel objet tandis que l'opérateur += modifie l'objet initial.

```
In [38]: liste1 = [1, 2, 3]

In [39]: liste2 = liste1

In [40]: liste1 = liste1 + [4, 5]

In [41]: liste1, liste2          # seule liste1 a ete modifiée
Out[41]: ([1, 2, 3, 4, 5], [1, 2, 3])

In [42]: id(liste1), id(liste2)  # c'est normal : liste1 et liste2 pointent vers des␣
↪objets distincts
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[42]: (140367336861256, 140367337087560)
```

```
In [43]: liste1 = [1, 2, 3]

In [44]: liste2 = liste1

In [45]: liste1 += [4, 5]

In [46]: liste1, liste2          # liste1 et liste2 ont ete modifiées
Out[46]: ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5])

In [47]: id(liste1), id(liste2)  # c'est normal : liste1 et liste2 pointent vers le␣
↪même objet
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[47]: (140367336874760,␣
↪140367336874760)
```

Egalité structurelle ou physique

On a vu que l'opérateur == permettait de tester si deux objets étaient égaux. Mais de quel type d'égalité parle-t-on alors ? L'opérateur == teste si deux objets ont la même **valeur** sans pour autant qu'il partage le même emplacement en mémoire. On parle alors d'**égalité structurelle**.

Lorsque « deux » objets sont en fait identiques (c'est-à-dire lorsqu'ils ont le même emplacement en mémoire), on parle d'**égalité physique**. Pour tester l'égalité physique, on peut comparer les emplacements en mémoire à l'aide de la fonction `id` ou plus simplement utiliser l'opérateur `is`.

```
In [48]: liste1 = [1, 2, 3]

In [49]: liste2 = liste1

In [50]: liste3 = liste1[:]

In [51]: liste1, liste2, liste3
Out[51]: ([1, 2, 3], [1, 2, 3], [1, 2, 3])

In [52]: id(liste1), id(liste2), id(liste3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[52]: (140367336373192, 140367336373192,
↪ 140367336889672)

In [53]: liste2 == liste1, liste3 == liste1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪(True, True)
```

```
In [54]: liste2 is liste1, liste3 is liste1
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪(True, False)
```

Un exemple peut-être un peu plus surprenant.

```
In [55]: [1, 2, 3] == [1, 2, 3]
Out[55]: True

In [56]: [1, 2, 3] is [1, 2, 3]
\\\\\\\\\\\\\\Out[56]: False
```

Python a en fait stocké deux versions de la liste `[1, 2, 3]` dans deux emplacements en mémoire distincts.

On termine par un cas plus vicieux que les deux exemples initiaux et qui peut faire passer des nuits blanches au programmeur débutant en Python.

```
In [57]: a = [[0] * 3] * 4

In [58]: a
Out[58]: [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

In [59]: a[0][0] = 1       # on pense n'avoir modifié qu'un élément de la liste de␣
↪listes a

In [60]: a                 # en fait non...
Out[60]: [[1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

Notes

3.5 Dictionnaires

CHAPITRE 4

Rudiments d'algorithmique

Jusqu'à maintenant, les exemples donnés dans le langage Python consistaient en une ou quelques instructions. Bien évidemment, si on veut créer des programmes plus élaborés, on ne peut pas se contenter d'une simple succession d'instructions. Par exemple, pour une tâche aussi simple que déterminer le plus grand élément d'une liste de nombres de longueur arbitraire, il faut pouvoir parcourir tous les éléments de la liste et choisir le maximum après avoir comparé les éléments les uns aux autres. Pour cela, il faut introduire ce qu'on appelle des *structures de contrôle*. Il en existe de plusieurs sortes :

— les tests (exécution d'un bloc d'instructions si une condition est remplie) ;
— les boucles (répétition d'un bloc d'instructions) ;
— les appels de fonctions (exécution d'un bloc d'instructions prédéfini).

4.1 Tests

Un test permet d'effectuer un bloc d'instructions lorsqu'une condition est remplie. On emploie pour cela le mot-clé `if` suivie d'une expression à valeur booléenne.

```
In [1]: a = 4        # Changer la valeur de a pour comprendre ce qui se passe

In [2]: if a % 2 ==0 :
   ...:     print('a est pair')
   ...:
a est pair
```

Si on veut introduire un bloc d'instructions à exécuter lorsque la condition **n'est pas** remplie, on emploie le mot-clé `else`.

```
In [3]: a = 5

In [4]: if a % 2 == 0:
   ...: ...  print('a est pair')
   ...: else:
   ...: ...  print('a est impair')
```

```
   ...:
a est impair
```

Si on veut envisager plusieurs tests successifs, on emploie le mot-clé `elif` (conctraction de `else if`).

```
In [5]: a = 5

In [6]: if a % 2 == 0:
   ...: ...  print('a est pair')
   ...: elif a > 3:
   ...: ...  print('a est impair et strictement supérieur à 3')
   ...:
a est impair et strictement supérieur à 3
```

On peut évidemment combiner `if`, `elif` et `else`.

```
In [7]: a = 1

In [8]: if a % 2 == 0:
   ...: ...  print('a est pair')
   ...: elif a > 3:
   ...: ...  print('a est impair et strictement supérieur à 3')
   ...: else:
   ...: ...  print('a est impair et inférieur ou égal à 3')
   ...:
a est impair et inférieur ou égal à 3
```

4.1.1 Opérateur ternaire `... if ... else ...`

Python dispose

```
<expression1> if <condition> else <expression2>
```

Cette expression est évaluée comme `<expression1>` si `<condition>` est vraie et comme `<expression2>` sinon.

```
In [9]: a = 0

In [10]: 'papa' if a == 0 else 'maman'
Out[10]: 'papa'

In [11]: 'papa' if a == 1 else 'maman'
\\\\\\\\\\\\\\\Out[11]: 'maman'
```

Ce type d'expression peut également accomplir une action plutôt que de renvoyer un objet.

```
In [12]: li = [1, 2, 3]

In [13]: a = 0

In [14]: li.append('toto') if a == 0 else li.append('titi')

In [15]: li
Out[15]: [1, 2, 3, 'toto']

# En fait, l'expression est évaluée à None
```

```
In [16]: print(li.append('toto') if a == 1 else li.append('titi'))
\\\\\\\\\\\\\\\\\\\\\\\\\\None

In [17]: li
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[17]: [1, 2, 3, 'toto', 'titi']
```

4.2 Boucles

Comme dans beau de langages de programmation, il existe deux types de boucles en Python :

- les boucles inconditionnelles qui permettent d'exécuter un bloc d'instructions un nombre de fois fixé à l'avance ;
- les boucles conditionnelles qui permettent l'exécution d'un code tant qu'une condition est remplie.

4.2.1 Boucles inconditionnelles

Les boucles inconditionnelles en Python permettent de parcourir un objet de type **itérable** (comme une liste, un tuple ou une chaîne de caractères) élément par élément.

De manière générale, on utilise les mots-clés `for` et `in`.

```
for <element> in <iterable>:
    <instruction1>
    <instruction2>
    ...
```

On peut itérer aussi bien sur des listes

```
In [1]: for elt in [1, 'toto', 2.34]:
   ...:     print(elt)
   ...:
1
toto
2.34
```

que sur des tuples

```
In [2]: for elt in 5, 6, 'tata':    # pas besoin de parenthèses pour le tuple dans ce␣
↪cas
   ...:     print(elt)
   ...:
5
6
tata
```

et même sur des chaînes de caractère.

```
In [3]: for elt in 'blabla':
   ...:     print(elt)
   ...:
b
l
a
b
```

```
l
a
```

La fonction `range`

La fonction `range` renvoie un itérable contenant des entiers. Plus précisément, lorsque `a, b, c` sont des entiers (`c!=0`),

— `range(a)` contient les entiers `0, 1, 2, ..., a-2, a-1` (aucun si `a <= 0`);

```
In [4]: for i in range(5):
   ...:     print(i)
   ...:
0
1
2
3
4
```

```
In [5]: for i in range(-5):
   ...:     print(i)
   ...:
```

— `range(a, b)` contient les entiers `a, a+1, a+2, ..., b-2, b-1` (aucun si `b <= a`);

```
In [6]: for i in range(3, 8):
   ...:     print(i)
   ...:
3
4
5
6
7
```

```
In [7]: for i in range(8, 3):
   ...:     print(i)
   ...:
```

— `range(a, b, c)` contient les entiers `a, a+c, a+2c, ...` jusqu'à atteindre `b` exclu (aucun si `c*(b-a)<=0`).

```
In [8]: for i in range(4, 9, 2):
   ...:     print(i)
   ...:
4
6
8
```

```
In [9]: for i in range(9, 4, 2):
   ...:     print(i)
   ...:
```

```
In [10]: for i in range(9, 4, -2):
   ....:     print(i)
   ....:
9
```

```
7
5
```

```
In [11]: for i in range(4, 9, -2):
   ....:     print(i)
   ....:
```

4.2.2 Listes en compréhension

En mathématiques, il existe plusieurs manières de décrire un même ensemble. L'ensemble $\mathcal{A}$ des entiers pairs compris entre 0 et 19 peut être défini en *extension* :

$$\mathcal{A} = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$$

Il peut également être décrit en *compréhension* :

$$\mathcal{A} = \{2n,\ n \in [\![0, 9]\!]\}$$

De la même manière, la liste de ces entiers peut être défini en Python en extension :

```
In [12]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Out[12]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

et en compréhension :

```
In [13]: [2*n for n in range(10)]
Out[13]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

On parle alors de *liste en compréhension*.

Une autre manière de définir $\mathcal{A}$ en compréhension est la suivante :

$$\mathcal{A} = \{x \in [\![0, 19]\!],\ x \equiv 0[2]\}$$

La version correspondante en Python est :

```
In [14]: [n for n in range(20) if n%2==0]
Out[14]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Bien entendu, on peut utiliser ce type de liste pour d'autres objets que des entiers [1].

```
In [15]: [s.upper() for s in ('toto', 'tata', 'titi', 'zozo', 'zaza', 'zizi') if␣
↪s[0]=='t']
Out[15]: ['TOTO', 'TATA', 'TITI']
```

[1] Les listes en compréhension peuvent être utilisées pour effectuer des actions plutôt que de calculer des valeurs.

```
In [16]: s = ([], [1, 2], ['titi', 'tata'])

In [17]: [li.append('toto') for li in s]
Out[17]: [None, None, None]

In [18]: s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]: (['toto'], [1, 2, 'toto'], ['titi', 'tata', 'toto'])
```

4.2.3 Boucles conditionnelles

Une boucle conditionnelle consiste à répéter un bloc d'instructions **tant qu'une condition est vraie**.

Notes

4.3 Fonctions

Une fonction est un bloc d'instructions que l'on peut appeler à tout moment d'un programme. Les fonctions ont plusieurs intérêts, notamment :

— la réutilisation du code : éviter de répéter les mêmes séries d'instructions à plusieurs endroits d'un programme ;
— la modularité : découper une tâche complexe en plusieurs sous-tâches plus simples.

4.3.1 Définir une fonction

Au cours des chapitres précédents, on a déjà rencontré de nombreuses fonctions telles que `print` ou `len`. Chacune de ces fonctions reçoit un argument et effectue une action (la fonction `print` affiche un objet à l'écran) ou renvoie une valeur (la fonction `len` renvoie la taille d'un itérable).

Jusqu'à maintenant, on s'est contenté de faire appel à des fonctions prédéfinies. Mais on peut également définir ses propres fonctions : il faut alors **déclarer** ces fonctions avant de les utiliser. De manière générale, la syntaxe d'une déclaration de fonctions est la suivante.

```
def <nom_fonction>(<paramètres>):    # En-tête de la fonction
    <instruction1>
    <instruction2>                   # Corps de la fonction
    ...
    return <valeur>
```

On décrit dans le *corps* de la fonction les traitements à effectuer sur les paramètres et on spécifie la valeur que doit renvoyer la fonction.

Considérons l'exemple simple suivant.

```
In [1]: def factorielle(n):
   ...:     a = 1
   ...:     for k in range(1, n+1):
   ...:         a *= k
   ...:     return a
   ...:
```

La fonction `factorielle` prend en argument un objet `n` (que l'on supposera être un entier naturel), calcule la factorielle de `n` à l'aide d'une variable `a` et renvoie cette valeur.

On constate que rien ne se passe lorsque la fonction est déclarée. Il faut **appeler** la fonction en fournissant une valeur à l'entier `n` pour que le code soit exécuté.

```
In [2]: factorielle(5)
Out[2]: 120

In [3]: factorielle(7)
\\\\\\\\\\\Out[3]: 5040
```

Note : Il faut bien faire la différence entre **la déclaration** et **l'appel** de la fonction. Lorsqu'une fonction est **déclarée**, aucun code n'est exécuté. Il faut **appeler** la fonction pour que le code soit exécuté.

4.3.2 L'instruction `return`

On « sort » de la fonction dès qu'on recontre une instruction `return` : en particulier, les instructions suivant un `return` ne sont pas exécutées.

```
In [4]: def test(n):
   ...:     if n % 2 == 0:
   ...:         return "n est un multiple de 2"
   ...:     if n % 3 ==0:
   ...:         return "n est un multiple de 3"
   ...:     return "Bidon"
   ...:

In [5]: test(4)
Out[5]: 'n est un multiple de 2'

In [6]: test(9)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]: 'n est un multiple de 3'

In [7]: test(6)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: 'n est un␣
↪multiple de 2'

In [8]: test(11)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪'Bidon'
```

On peut cependant utiliser ceci à notre avantage : par exemple, pour sortir d'une boucle `for` avant d'avoir accompli toutes les itérations.

```
In [9]: from math import sqrt, floor

In [10]: def est_premier(n):
   ....:     if n <= 1:
   ....:         return False
   ....:     for d in range(2, floor(sqrt(n)) + 1):
   ....:         if n % d == 0:
   ....:             return False
   ....:     return True
   ....:

In [11]: print([(n, est_premier(n)) for n in range(10)])
[(0, False), (1, False), (2, True), (3, True), (4, False), (5, True), (6, False), (7,␣
↪True), (8, False), (9, False)]
```

Une fonction peut ne pas contenir d'instruction `return` ou peut ne renvoyer aucune valeur. En fait, si on ne renvoie pas explicitement de valeur, Python renverra par défaut la valeur particulière `None`.

```
In [12]: def f(x):
   ....:     x**2
   ....:
```

```
In [13]: print(f(2))
None
```

```
In [14]: def g(x):
   ....:     2 * x
   ....:     return
   ....:
```

```
In [15]: print(g(2))
None
```

Avertissement : Une erreur de débutant consiste à confondre les utilisations de `print` et `return` : une fonction ne compotant qu'un `print` et pas de `return` ne fera qu'afficher un résultat à l'écran mais ne renverra aucune valeur.

```
In [16]: def bidon():
   ....:     print(1)
   ....:     return 2
   ....:

In [17]: a = bidon() # La fonction bidon affiche bien 1
1

In [18]: a           # Mais elle a renvoyé la valeur 2
\\Out[18]: 2
```

La plupart du temps, on préfèrera utiliser utiliser `return` plutôt que `print` : l'objet affiché par `print` est en quelque sorte « perdu » pour le reste du programme s'il n'a pas été renvoyé via `return`.

```
In [19]: def liste_carres1(n):
   ....:     print([k**2 for k in range(1, n+1)])
   ....:
```

```
In [20]: def liste_carres2(n):
   ....:     return [k**2 for k in range(1, n+1)]
   ....:
```

```
# Avec la première version, la liste des carrés est affichée mais on ne peut plus␣
↪rien en faire
In [21]: liste_carres1(10)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# En effet, la fonction renvoie None
In [22]: print(liste_carres1(10))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
None

# Avec la deuxième version, on peut par exemple calculer la somme des carrés des␣
↪premiers entiers
In [23]: sum(liste_carres2(10))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]:␣
↪385
```

4.3.3 Paramètres et arguments

Une fonction peut avoir zéro, un ou plusieurs paramètres [1].

Note : Bien que les termes *paramètres* et *arguments* soient souvent confondus, il existe une nuance dont nous tiendrons compte dans ce chapitre : les *paramètres* sont les noms intervenant dans l'en-tête de la fonction tandis que les *arguments* sont les valeurs passées à la fonction lors de son appel.

```
In [24]: def add(a, b):       # Les paramètres sont a et b
   ....:     return a + b
   ....:

In [25]: add(5, 10)           # Les arguments sont 5 et 10
Out[25]: 15
```

De même que pour les variables, les noms des paramètres doivent refléter leur utilisation pour que le code soit plus lisible. Par ailleurs, on peut passer des arguments à une fonction en utilisant les noms des paramètres, ce qui rend le code encore plus explicite.

```
In [26]: def nom_complet(prenom, nom):
   ....:     return prenom[0].upper() + prenom[1:].lower() + ' ' + nom.upper()
   ....:

In [27]: nom_complet(prenom='james', nom='bond')
Out[27]: 'James BOND'
```

L'emploi d'*arguments nommés* permet de passer les arguments dans un ordre différent de l'ordre des paramètres dans l'en-tête de la fonction.

```
In [28]: nom_complet(nom='PrOuSt', prenom='MARcel')
Out[28]: 'Marcel PROUST'
```

Il est possible de donner des valeurs par défaut aux paramètres d'une fonction : les arguments correspondants ne sont plus alors requis lors de l'appel de la fonction.

```
In [29]: def nom_complet(prenom='Joe', nom='Bob'):
   ....:     return prenom[0].upper() + prenom[1:].lower() + ' ' + nom.upper()
   ....:

In [30]: nom_complet()
Out[30]: 'Joe BOB'

In [31]: nom_complet('ulysse')
\\\\\\\\\\\\\\\\\\Out[31]: 'Ulysse BOB'

In [32]: nom_complet(nom='capet')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[32]: 'Joe CAPET'
```

Une fonction sans paramètre nécessite quand même des parenthèses dans son en-tête.

```
In [92]: def f():
   ....:     return 1
   ....:

In [93]: f()
Out[93]: 1
```

Dans l'en-tête d'une fonction les paramètres avec des valeurs par défaut doivent toujours *suivre* les paramètres sans valeurs par défaut sous peine de déclencher une erreur de syntaxe.

```
In [33]: def toto(a=1, b, c=2):
   ....:     pass
   ....:
  File "<ipython-input-33-ff9b54c14016>", line 1
    def toto(a=1, b, c=2):
             ^
SyntaxError: non-default argument follows default argument
```

Le but est d'éviter toute ambiguïté. En effet, quels seraient les arguments passés lors de l'appel de fonction `toto(5, 6)` ? `a=1`, `b=5` et `c=6` ou bien `a=5`, `b=6` et `c=2` ?

4.3.4 Portée des variables

Une fonction peut utiliser des variables définies **à l'extérieur** de cette fonction.

```
In [34]: a = 2

In [35]: def f(x):
   ....:     return a * x
   ....:

In [36]: f(5)
Out[36]: 10
```

On dit que les variables définies à l'extérieur d'une fonction sont des variables **globales**.

Note : De manière générale, il est plutôt déconseillé d'utiliser des variables globales à l'intérieur d'une fonction. Il est par exemple plus difficile de tester ou débugger une fonction faisant appel à des variables globales : en plus de chercher les bugs à l'intérieur de la fonction, il faudra examiner tous les endroits où ces variables globales sont potentiellement modifiées, ce qui peut devenir un vrai casse-tête dans un programme complexe.

Avertissement : Si on veut utiliser une variable globale à l'intérieur d'une fonction, il faut que celle-ci soit déclarée **avant** l'appel de cette fonction.

```
In [37]: def f(x):
   ....:     return b * x
   ....:

In [38]: f(5)
Out[38]: 35

In [39]: b = 2
```

Considérons maintenant l'exemple suivant.

```
In [40]: a = 1

In [41]: def f():
   ....:     a = 2
   ....:     return None
```

```
   ....:

In [42]: a
Out[42]: 1

In [43]: f()

In [44]: a   # a vaut toujours 1
Out[44]: 1
```

On dit que les variables à l'intérieur d'une fonction sont des variables **locales**. Cela signifie en particulier que des opérations effectuées sur une variable d'un certain nom **à l'intérieur** d'une fonction ne modifient pas une variable du même nom **à l'extérieur** de cette fonction.

Note : On évitera cependant de donner des noms identiques à des variables locales et globales de manière à éviter toute confusion.

Quand il existe des variables locales et globales de même nom, la préférence est donnée aux variables locales à l'intérieur de la fonction.

```
In [45]: a = 1

In [46]: def f(x):
   ....:     a = 3
   ....:     return a + x    # la variable locale a est utilisée et non la variable␣
↪globale a
   ....:

In [47]: f(5)
Out[47]: 8
```

On ne peut pas accéder à des variables locales à l'extérieur de la fonction où elles sont définies.

```
In [48]: def f():
   ....:     c = 2
   ....:     return None
   ....:

In [49]: f()

In [50]: c   # c est inconnu à l'extérieur de la fonction
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-50-e81f39fb0b48> in <module>()
----> 1 c   # c est inconnu à l'extérieur de la fonction

NameError: name 'c' is not defined
```

On peut donc également voir les variables locales comme des variables *temporaires* dont l'existence n'est assurée qu'à l'intérieur de la fonction où elles interviennent.

On peut néanmoins modifier une variable globale à l'intérieur d'une fonction : on utilise alors le mot-clé `global`.

```
In [51]: a = 1

In [52]: def f():
```

```
   ....:     global a
   ....:     a = 2
   ....:     return None
   ....:

In [53]: a
Out[53]: 1

In [54]: f()

In [55]: a   # a vaut bien 2
Out[55]: 2
```

Les paramètres d'une fonction ont également une portée locale.

```
In [56]: def f(c):
   ....:     return 2 * c
   ....:

In [57]: c   # c est inconnu à l'extérieur de la fonction
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-57-e81f39fb0b48> in <module>()
----> 1 c   # c est inconnu à l'extérieur de la fonction

NameError: name 'c' is not defined
```

4.3.5 Fonctions et mutabilité

Considérons ce premier exemple où l'argument est un entier.

```
In [58]: def f(x):
   ....:     x += 1
   ....:
```

```
In [59]: a = 2

In [60]: f(a)

In [61]: a           # la variable a n'est pas modifiée
Out[61]: 2
```

Et maintenant, un deuxième exemple où l'argument est une liste.

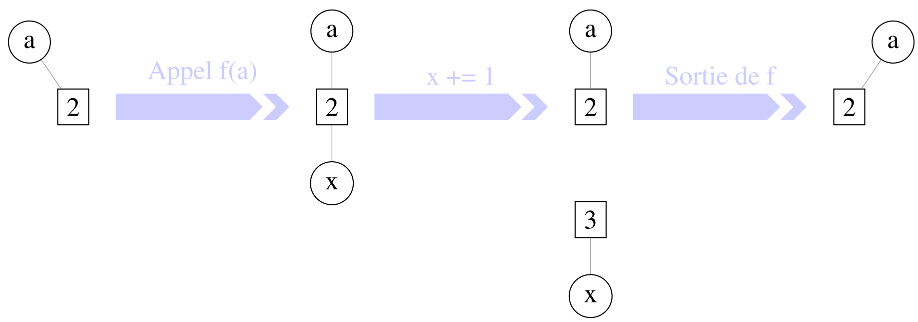
```
In [62]: def g(li):
   ....:     li.append(3)
   ....:
```

```
In [63]: lst = [1, 2]

In [64]: g(lst)

In [65]: lst       # la variable lst a été modifiée
Out[65]: [1, 2, 3]
```

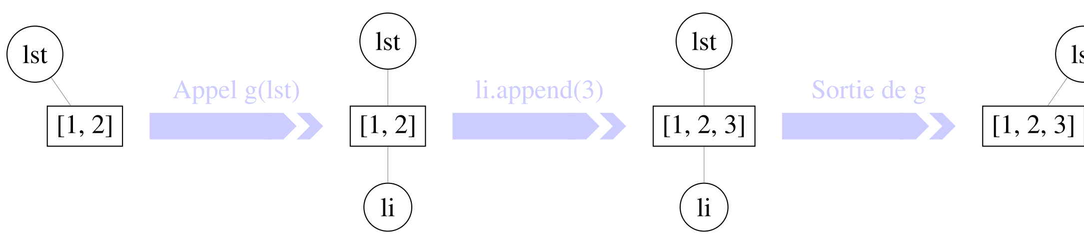
Le résultat du deuxième exemple peut sembler étrange puisqu'une variable globale a été modifiée à l'intérieur d'une fonction. Pour expliquer cette différence de comportement, il faut comprendre plus en détail comment sont passés les arguments à une fonction et faire une distinction entre les objets *mutables* et *immutables*.

— Lors de l'exécution des instructions `f(a)` et `g(lst)`, les emplacements en mémoire dans lesquels sont stockés les objets associés aux variables `a` et `lst` (c'est-à-dire l'entier `2` et la lst `[1, 2]`) sont passés aux fonctions `f` et `g` et les paramètres `x` et `li` pointent alors vers ces emplacements en mémoire.

— Puisqu'un entier est un objet immutable, l'instruction `x += 1` fait pointer le paramètre `x` vers un nouvel emplacement mémoire où est stocké l'entier `3`. Cependant, a variable `a` pointe toujours vers l'ancien emplacement en mémoire et est donc toujours associée à l'entier `2`.



— Par contre, une liste étant un objet mutable, l'instruction `li.append(3)` modifie l'objet stocké à l'emplacement en mémoire vers lequel pointe `li`. Cet objet vaut alors `[1, 2, 3]`. Mais la variable `lst` pointe toujours vers le même emplacement en mémoire et est donc associé à cet objet modifié.



On se convaincra plus facilement en utilisant la fonction `id` qui renvoie l'emplacement où est stocké un objet en mémoire et l'opérateur `is` qui teste si deux objets sont physiquement égaux (c'est-à-dire s'ils occupent le même emplacement en mémoire).

```
In [66]: def f(x):
    ....:     print('x début fonction f', id(x), x is a)
    ....:     x += 1
    ....:     print('x fin fonction f', id(x), x is a)
    ....:
```

```
In [67]: a = 2

In [68]: print('a avant appel fonction f', id(a))
a avant appel fonction f 10911200

In [69]: f(a)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\x début fonction f 10911200 True
x fin fonction f 10911232 False

In [70]: print('a après appel fonction f', id(a))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\a
↪après appel fonction f 10911200

In [71]: a
```

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪2
```

```
In [72]: def g(li):
   ....:     print('li début fonction g', id(li), lst is li)
   ....:     li.append('toto')
   ....:     print('li fin fonction g', id(li), lst is li)
   ....:
```

```
In [73]: lst = [1, 2, 3]

In [74]: print('lst avant appel fonction g', id(lst))
lst avant appel fonction g 140367337678856

In [75]: g(lst)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\li début fonction g 140367337678856 True
li fin fonction g 140367337678856 True

In [76]: print('lst après appel fonction g', id(lst))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪après appel fonction g 140367337678856

In [77]: lst
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[1, 2, 3, 'toto']
```

Finalement, on peut résumer les choses de la manière suivante.

Astuce : Un objet mutable peut-être modifé s'il est passé en argument à une fonction alors que ce ne sera jamais le cas pour un objet immutable.

4.3.6 Effets de bord

En plus de renvoyer une valeur, une fonction peut entraîner des modifications au-delà de sa portée comme :
- modifier des variables globales ;
- modifier des arguments mutables ;
- afficher des informations à l'écran ;
- enregistrer des données dans un fichier.

On parle alors d'**effet de bord**. Les effets de bord sont à utiliser avec parcimonie : en effet,

4.3.7 Une fonction est un objet

Il est important de noter qu'en Python, les fonctions sont des objets commes les autres (entiers, tuples, ...). Notamment, une fonction possède un type.

```
In [78]: def f(x):
   ....:     return 2 * x
   ....:

In [79]: type(f)
Out[79]: function
```

Ceci est important car on peut par exemple utiliser une fonction comme un argument d'une autre fonction.

```
In [80]: def appliquer(f, x):
   ....:     return f(x)
   ....:

In [81]: def f(x):
   ....:     return 2 * x
   ....:

In [82]: appliquer(f, 5)
Out[82]: 10
```

On peut également créer une fonction qui renvoie une autre fonction.

```
In [83]: def multiplier_par(a):
   ....:     def f(x):
   ....:         return a * x
   ....:     return f
   ....:

In [84]: multiplier_par(2)(5)
Out[84]: 10
```

4.3.8 Fonctions anonymes

En mathématiques, on peut parler d'une fonction de plusieurs manières.

— On peut lui donner un nom : on peut par exemple considérer la fonction f telle que $f(x) = x^2$.

— Mais si on ne compte pas réutiliser plus tard cette fonction, on peut tout simplement parler de la fonction $x \mapsto x^2$.

De la même manière, on peut nommer explicitement une fonction.

```
def f(x):
    return x**2
```

On peut également utiliser une *fonction anonyme* (également appelée *fonction lambda*).

```
lambda x: x**2
```

```
In [85]: (lambda x: x**2)(4)
Out[85]: 16

In [86]: f = lambda x: x**2              # On peut bien sûr donner un nom à une␣
↪fonction anonyme

In [87]: f(4)
Out[87]: 16

In [88]: g = lambda x, y: x**2 + y**2    # Une fonction anonyme peut avoir plus d'un␣
↪argument

In [89]: g(1, 2)
Out[89]: 5
```

De manière générale, la syntaxe d'une fonction anonyme est la suivante.

```
lambda <paramètres>: <expression>
```

A la différence d'une fonction classique, une fonction anonyme ne nécessite pas d'instruction `return` : l'expression suivant `:` est renvoyée [2].

Les fonctions anonymes sont limitées par rapport aux fonctions classiques : elles ne peuvent pas exécuter plusieurs instructions puisque seule **une** expression est renvoyée. Quel est alors l'intérêt des fonctions anonymes ? Il s'agit de créer des fonctions à usage unique qui peuvent notamment servir d'arguments dans d'autres fonctions.

Par exemple, Python dispose d'une fonction `map` qui permet d'appliquer une fonction à chaque élément d'un objet de type itérable.

```
In [90]: list(map(lambda x: 2*x, [1, 2, 3]))    # la fonction map renvoie un objet de␣
↪type map qu'on convertit en liste
Out[90]: [2, 4, 6]
```

Bien entendu, on arriverait plus aisément au même résultat grâce à une liste en compréhension.

```
In [91]: [2 * x for x in [1, 2, 3]]
Out[91]: [2, 4, 6]
```

Notes

[2] Une fonction anonyme peut également avoir des effets de bord.

```
In [94]: f = lambda li: li.append('toto')

In [95]: a = [1, 2]

In [96]: f(a)

In [97]: a                # La fonction anonyme f a modifié la liste a
Out[97]: [1, 2, 'toto']

In [98]: print(f(a))      # Par contre, la fonction ne renvoie rien (en fait, renvoie None)
\\\\\\\\\\\\\\\\\\\\\\\\None
```

CHAPITRE 5

Algorithmes classiques

On détaille dans ce chapitre quelques algorithmes classiques dont certains figurent officiellement au programme de CPGE.

5.1 Algorithmes de recherche

5.1.1 Recherche d'un élément dans une liste

Il faut noter que Python dispose déjà de l'opérateur `in` pour tester si un élément figure dans une liste.

```
In [1]: 2 in [5, 4, 1, 2, 3]
Out[1]: True

In [2]: 6 in [5, 4, 1, 2, 3]
\\\\\\\\\\\\Out[2]: False
```

La méthode `index` permet de renvoyer l'indice de l'élément dans la liste s'il a été trouvé.

```
In [3]: [5, 4, 1, 2, 3].index(2)
Out[3]: 3

In [4]: [5, 4, 1, 2, 3].index(6)
\\\\\\\\\\----------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-b8fd51641ee1> in <module>()
----> 1 [5, 4, 1, 2, 3].index(6)

ValueError: 6 is not in list
```

On peut néanmoins proposer notre propre algorithme : il suffit de balayer la liste et de renvoyer `True` dès qu'on trouve l'élément recherché et `False` si on a parcouru toute la liste sans trouver l'élément.

```
In [5]: def appartient(elt, lst):
   ...:     for e in lst:
   ...:         if e == elt:
   ...:             return True
   ...:     return False
   ...:

In [6]: appartient(2, [5, 4, 1, 2, 3])
Out[6]: True

In [7]: appartient(6, [5, 4, 1, 2, 3])
\\\\\\\\\\\\Out[7]: False
```

On peut également proposer une version qui renvoie l'indice la première occurence de l'élément recherché s'il est trouvé et `None` sinon.

```
In [8]: def indice(elt, lst):
   ...:     for i in range(len(lst)):
   ...:         if lst[i] == elt:
   ...:             return i
   ...:     return None
   ...:

In [9]: indice(2, [5, 4, 1, 2, 3])
Out[9]: 3

In [10]: indice(6, [5, 4, 1, 2, 3])  # L'interpréteur IPython n'affiche pas None
```

5.1.2 Recherche d'un élément dans une liste triée

Lorsque l'on dispose d'une liste triée par ordre croissant, on peut grandement améliorer notre algorithme en utilisant le principe de **dichotomie**.

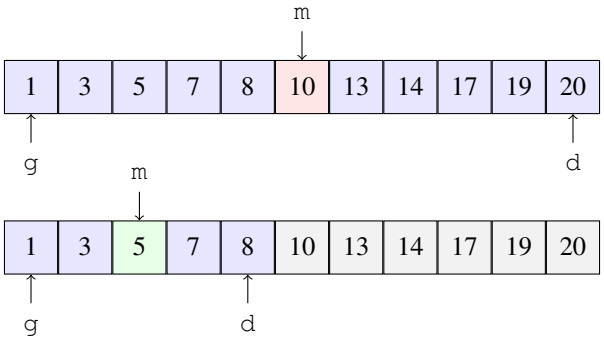
- On recherche tout d'abord l'élément central de la liste.
- Si c'est l'élément recherché, on s'arrête. Sinon, on le compare à l'élément recherché.
- Si l'élément recherché est inférieur à l'élément central, on le recherche dans la première partie de la liste. Sinon, on le recherche dans la deuxième partie de la liste.
- On retourne donc à la première étape de notre algorithme appliqué à l'une des deux demi-listes.

```
In [11]: def appartient_dicho(elt, lst):
   ....:     g = 0
   ....:     d = len(lst) - 1
   ....:     while g <= d:
   ....:         m = (g + d) // 2
   ....:         if lst[m] == elt:
   ....:             return True
   ....:         if elt < lst[m]:
   ....:             d = m - 1
   ....:         else:
   ....:             g = m + 1
   ....:     return False
   ....:

In [12]: appartient_dicho(13, [1, 3, 5, 7, 8, 10, 13, 14, 17, 19])
Out[12]: True
```

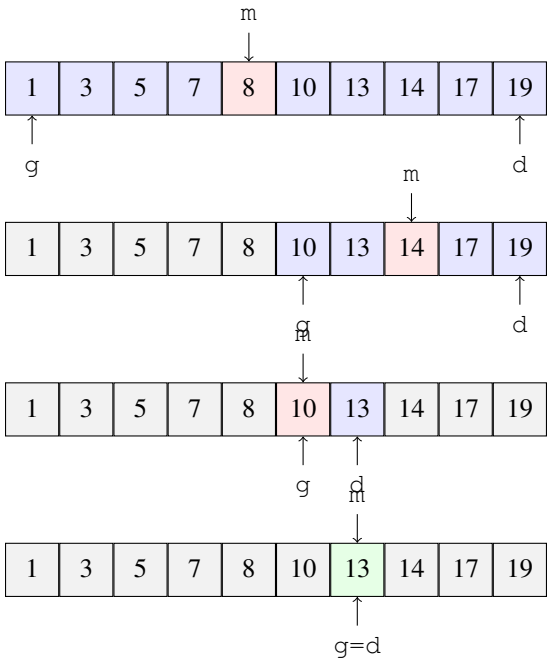
```
In [13]: appartient_dicho(18, [1, 3, 5, 7, 8, 10, 13, 14, 17, 19])
\\\\\\\\\\\\\Out[13]: False
```

Comme souvent, un dessin vaut mieux qu'un long discours. On donne deux exemples d'application de cet algorithme.

Recherche de `5` dans la liste `[1, 3, 5, 7, 8, 10, 13, 14, 17, 19, 20]`



Recherche de `13` dans la liste `[1, 3, 5, 7, 8, 10, 13, 14, 17, 19]`



A nouveau, on peut proposer une version qui renvoie l'indice de la première occurence de l'élément recherché plutôt qu'un booléen.

```
In [14]: def indice_dicho(elt, lst):
   ....:     g = 0
   ....:     d = len(lst) - 1
   ....:     while g <= d:
   ....:         m = (g + d) // 2
   ....:         if lst[m] == elt:
```

```
   ....:             return m
   ....:         if elt < lst[m]:
   ....:             d = m - 1
   ....:         else:
   ....:             g = m + 1
   ....:     return None
   ....:

In [15]: indice_dicho(13, [1, 3, 7, 8, 10, 13, 14, 17, 19])
Out[15]: 5

In [16]: indice_dicho(18, [1, 3, 7, 8, 10, 13, 14, 17, 19])  # L'interpréteur IPython␣
↪n'affiche pas None
```

Comparaison de l'efficacité des deux algorithmes

On peut comparer les temps de calcul des deux versions de l'algorithme de recherche d'un élément grâce à la *commande magique* `%timeit` : celle-ci permet d'exécuter un grand nombre de fois la même instruction et de mesurer le temps d'exécution moyen de cette instruction.

On remarque en particulier que le temps de calcul avec l'algorithme standard augmente à peu près proportionnellement à la taille de la liste tandis que le temps de calcul avec l'algorithme par dichotomie augmente très peu avec la taille de la liste. Le gain de temps de calcul est donc d'autant plus grand que la liste est grande [1]. On donnera une comparaison quantitative de ces deux temps de calcul dans le chapitre sur la complexité.

5.1.3 Recherche du maximum ou du minimim d'une liste

On suppose qu'on dispose d'une liste d'éléments que l'on peut comparer les uns aux autres et on cherche à déterminer le plus grand ou le plus petit élément. Python dispose déjà de deux fonctions `min` et `max` pour effectuer cette tâche.

```
In [17]: lst = [5, -7, 4, -3, 2, 10]

In [18]: min(lst), max(lst)
Out[18]: (-7, 10)
```

On peut également proposer notre algorithme. Rien de bien difficile, il suffit de parcourir un à un les éléments de la liste et de comparer chaque élément au minimum ou maximum des éléments précédents.

```
In [19]: def minmax(lst):
   ....:     m, M = None, None
   ....:     for elt in lst:
   ....:         if m is None or m > elt:
   ....:             m = elt
   ....:         if M is None or M < elt:
   ....:             M = elt
   ....:     return m, M
   ....:

In [20]: minmax([5, -7, 4, -3, 2, 10])
Out[20]: (-7, 10)
```

[1] Il ne faut cependant pas crier tout de suite au miracle. L'algorithme de recherche par dichotomie nécessite que la liste traitée soit auparavant triée. Et le tri est une opération qui nécessite un certain temps de calcul (plus élevé que celui de l'algorithme de recherche standard).

5.1.4 Recherche d'une sous-chaîne dans une chaîne de caractères

L'objectif est de retrouver une sous-chaîne (qu'on appellera *motif*) dans une chaîne de caractères. Par exemple, la chaîne `"pitapipapa"` contient le motif `"pipa"` mais pas le motif `"tapi"`. Python propose déjà cette fonctionnalité à l'aide de l'opérateur `in`.

```
In [21]: "pipa" in "pitapipapa"
Out[21]: True

In [22]: "tapa" in "pitapipapa"
\\\\\\\\\\\\\\Out[22]: False
```

La méthode `index` permet de renvoyer l'indice du caractère où a été trouvé le motif le cas échéant.

```
In [23]: "pitapipapa".index("pipa")
Out[23]: 4

In [24]: "pitapipapa".index("tapa")
\\\\\\\\\\\\----------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-3adb3dc6f3c0> in <module>()
----> 1 "pitapipapa".index("tapa")

ValueError: substring not found
```

On présente ici un algorithme naïf qui est assez peu efficace mais qui a le mérite d'être très facile à comprendre : on prend successivement chaque caractère de la chaîne comme point de départ et on compare les caractères de la chaîne et les caractères du motif à partir de ce point de départ.

```
In [25]: def recherche_chaine(chaine, motif):
   ....:     n = len(chaine)
   ....:     m = len(motif)
   ....:     for ind in range(n-m):
   ....:         nb = 0
   ....:         while nb < m and chaine[ind+nb] == motif[nb]:
   ....:             nb +=1
   ....:         if nb == m:
   ....:             return True
   ....:     return False
   ....:

In [26]: recherche_chaine("pitapipapa", "pipa")
Out[26]: True

In [27]: recherche_chaine("patapipapa", "tapa")
\\\\\\\\\\\\\\Out[27]: False
```

Recherche du motif `"pipa"` dans la chaîne `"pitapipapa"`

p	i	t	a	p	i	p	a	p	a	ind=0, nb=1
p	i	t	a	p	i	p	a	p	a	ind=0, nb=2
p	i	t	a	p	i	p	a	p	a	ind=0, nb=2
p	**i**	t	a	p	i	p	a	p	a	ind=1, nb=0
p	i	**t**	a	p	i	p	a	p	a	ind=2, nb=0
p	i	t	**a**	p	i	p	a	p	a	ind=3, nb=0
p	i	t	a	**p**	i	p	a	p	a	ind=4, nb=1
p	i	t	a	**p**	i	p	a	p	a	ind=4, nb=2
p	i	t	a	**p**	i	p	a	p	a	ind=4, nb=3
p	i	t	a	**p**	i	p	a	p	a	ind=4, nb=4

On peut à nouveau proposer une version de l'algorithme qui renvoie l'indice de la *première occurence* rencontrée.

```
In [28]: def indice_chaine(chaine, motif):
    ....:     n = len(chaine)
    ....:     m = len(motif)
    ....:     for ind in range(n-m):
    ....:         nb = 0
    ....:         while nb < m and chaine[ind+nb] == motif[nb]:
    ....:             nb +=1
    ....:         if nb == m:
    ....:             return nb
    ....:     return None
    ....:

In [29]: indice_chaine("pitapipapa", "pipa")
Out[29]: 4

In [30]: indice_chaine("patapipapa", "tapa")      # L'interpréteur IPython n'affiche␣
↪pas None
```

Notes

5.2 Analyse numérique

5.2.1 Résolution d'équations par dichotomie

Il s'agit ici de calculer une *valeur approchée* d'une solution d'une équation du type $f(x) = 0$. On ne cherche pas à obtenir une expression exacte d'une telle solution, ce qui est de toute façon évidemment impossible de manière générale.

On suppose qu'on dispose d'une fonction f continue et strictement monotone sur un intervale $[a, b]$ vérifiant $f(a)f(b) \leq 0$. Le théorème des valeurs intermédiaires garantit l'existence d'une unique solution à l'équation $f(x) = 0$ sur l'intervalle $[a, b]$. Pour obtenir une *valeur approchée* de cette solution, on procède par **dichotomie** :

1. On calcule $c = (a + b)/2$ et $f(c)$.
2. Si $f(a)f(c) \leq 0$, la solution appartient à l'intervalle $[a, c]$. Sinon, elle appartient à l'intervalle $[c, b]$.
3. Dans le premier cas, on remplace b par c tandis que dans le second cas, on remplace a par c.

4. On répète les étapes 1., 2. et 3. tant que la longeur de l'intervalle $[a, b]$ est supérieur à une précision ϵ donnée.
5. La valeur de c est alors une valeur appochée de la solution de $f(x) = 0$ à $\epsilon/2$ près.

```
In [1]: def dicho(f, a, b, eps):
   ...:     while abs(b-a) > eps:
   ...:         c = (a + b) / 2
   ...:         if f(a) * f(c) <= 0:
   ...:             b = c
   ...:         else:
   ...:             a = c
   ...:     return (a+b) / 2
   ...:

In [2]: from math import sin

In [3]: dicho(sin, 2, 4, .0001)
Out[3]: 3.141571044921875
```

Il est à remarquer que le module `scipy` dispose déjà de fonctions pouvant résoudre de manière approchée des équations du type $f(x) = 0$.

```
In [4]: from scipy.optimize import fsolve

In [5]: from math import cos

In [6]: f = lambda x: cos(x**2)

In [7]: x0 = fsolve(f, 0)

In [8]: x0
Out[8]: array([-49.99136881])

In [9]: f(*x0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]: -1.063024370691732e-13
```

5.2.2 Calcul d'intégrales

On expose ici des algorithmes de calcul *approché* d'intégrales [1]. A nouveau, on ne cherche pas à obtenir des expressions exactes de ces intégrales.

Méthode des rectangles

On peut approcher une intégrale par une somme d'aire de rectangles comme l'indique la figure suivante.

[1] Le module `scipy.integrate` dispose déjà d'une fonction `quad` à cet effet.

```
In [53]: from scipy.integrate import quad

In [54]: quad(lambda x: 1 / x**2, 1, 2)
Out[54]: (0.4999999999999994, 5.551115123125782e-15)
```

La fonction `quad` renvoie un couple formé de l'approximation de l'intégrale et d'une majoration de l'erreur d'approximation.

Plus précisément, en posant $x_k = a + k(b-a)/n$ où n désigne le nombre de rectangles :

$$R_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f(x_k) \approx \int_a^b f(t)\,\mathrm{d}t \qquad \text{(rectangles verts)}$$

$$S_n = \frac{b-a}{n}\sum_{k=1}^{n} f(x_k) \approx \int_a^b f(t)\,\mathrm{d}t \qquad \text{(rectangles rouges)}$$

On peut alors proposer la fonction suivante pour approcher l'intégrale d'une fonction f sur un intervalle $[a, b]$.

```
In [10]: def rectangles(f, a, b, N, side):
   ....:     pas = (b-a) / N
   ....:     x = a
   ....:     somme = 0
   ....:     for _ in range(N):
   ....:         if side:
   ....:             somme += f(x)
   ....:         x += pas
   ....:         if not side:
   ....:             somme += f(x)
   ....:     return somme / N
   ....:

In [11]: rectangles(lambda x: x**2, 0, 1, 100, True)
Out[11]: 0.3283500000000004

In [12]: rectangles(lambda x: x**2, 0, 1, 100, False)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]: 0.33835000000000043
```

Les sommes R_n et S_n sont appelées des *sommes de Riemann* et on peut même prouver que pour une fonction f continue,

$$\lim_{n\to+\infty} S_n = \lim_{n\to+\infty} T_n = \int_a^b f(t)\,\mathrm{d}t$$

En particulier, l'appoximation de l'intégrale $\int_a^b f(t)\,\mathrm{d}t$ est d'autant meilleure que le nombre n de rectangles est grand, ce qui se conçoit très bien géométriquement [2].

Méthode des trapèzes

On peut également apporcher une intégrale comme une somme d'aires de trapèzes comme sur la figure suivante. Bien évidemment, l'approximation de l'intégrale est meilleure qu'avec des rectangles.

A nouveau, en posant $x_k = a + k(b-a)/n$ où n désigne le nombre de trapèzes :

$$T_n = \frac{b-a}{n}\sum_{k=0}^{n-1} \frac{f(x_k) + f(x_{k+1})}{2} \approx \int_a^b f(t)\,\mathrm{d}t$$

On peut évidemment remarquer que $T_n = (R_n + S_n)/2$. En fait, la somme précédente peut se réécrire de manière différente :

$$T_n = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(x_k)\right)$$

[2] Il ne faut cependant pas crier victoire trop tôt. Tout d'abord, le temps de calcul augmente avec n. De plus, chaque opération dans l'algorithme entraîne une erreur d'arrondi minime mais, le nombre d'opérations augmentant avec n, le cumul de ces erreurs d'arrondi finit par dépasser le gain en précision lorsque n est très grand.

Cette nouvelle formule permet de calculer T_n en effectuant moins d'opérations qu'avec la formule précédente. On peut alors donner l'algorithme suivant.

```
In [13]: def trapezes(f, a, b, N):
   ....:     pas = (b-a) / N
   ....:     x = a
   ....:     somme = (f(a) + f(b)) / 2
   ....:     for _ in range(N-1):
   ....:         x += pas
   ....:         somme += f(x)
   ....:     return somme / N
   ....:

In [14]: trapezes(lambda x: x**2, 0, 1, 100)
Out[14]: 0.3333500000000004
```

5.2.3 Résolution d'équations différentielles

L'objectif est de résoudre numériquement des équations différentielles : c'est-à-dire qu'on ne cherche pas des expressions explicites des solutions mais des valeurs approchées [3].

Pour commencer, on traitera le cas de *problème de Cauchy* d'ordre 1.

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases}$$

On rappelle qu'un tel problème consiste en la donnée d'une équation différentielle résolue d'ordre 1 $y' = f(t, y)$ et d'une condition initiale $y(t_0) = y_0$. Le théorème de Cauchy-Lipschitz garantit l'existence et l'unicité d'une solution à ce problème lorsque f est suffisamment règulière.

L'idée est d'utiliser une approximation affine de la fonction solution : $y(t + \Delta t) \approx y(t) + y'(t)\Delta t$. Le calcul de $y'(t)$ est possible grâce à l'équation différentielle si l'on connaît $y(t)$ puisque $y'(t) = f(t, y(t))$. On itère ce processus pour calculer des valeurs approchées à des intervalles de temps réguliers. Plus précisément, en posant $t_k = t_0 + k\Delta t$, on a alors

$$\begin{aligned} y(t_1) &\approx y(t_0) + y'(t_0)\Delta t = y(t_0) + f(t_0, y_0)\Delta t = y_1 \\ y(t_2) &\approx y(t_1) + y'(t_1)\Delta t \approx y(t_1) + f(t_1, y_1)\Delta t = y_2 \\ y(t_3) &\approx y(t_2) + y'(t_2)\Delta t \approx y(t_2) + f(t_2, y_2)\Delta t = y_3 \\ &\dots \end{aligned} \tag{5.4}$$

La méthode que l'on vient de décrire porte le nom de **méthode d'Euler**.

```
In [15]: def euler(f, t0, y0, pas, nb):
   ....:     t = t0
   ....:     y = y0
   ....:     liste_t = [t]
   ....:     liste_y = [y]
   ....:     for _ in range(nb):
   ....:         y += f(t, y) * pas
   ....:         t += pas
   ....:         liste_t.append(t)
   ....:         liste_y.append(y)
   ....:     return liste_t, liste_y
   ....:
```

[3] Le module `scipy.integrate` dispose déjà d'une fonction `odeint` à cet effet.

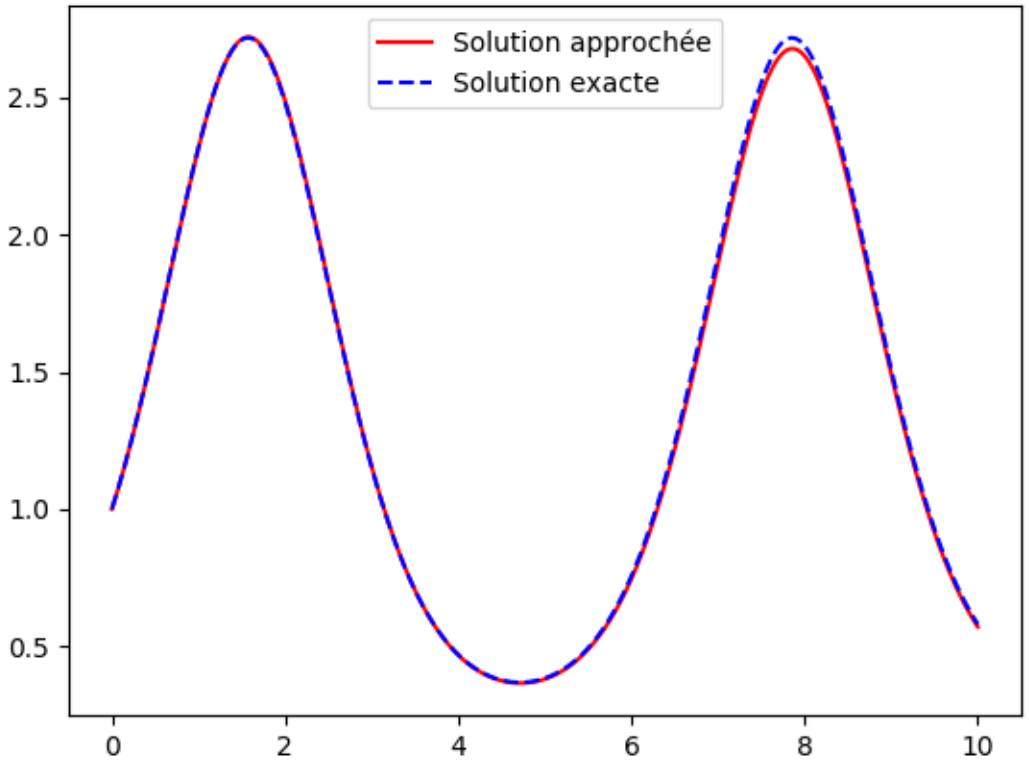
Par exemple, on calcule ici une solution approchée du système de Cauchy

$$\begin{cases} y' = \cos(t)y \\ y(0) = 1 \end{cases}$$

```
In [16]: from math import cos

In [17]: f = lambda t, y: cos(t) * y

In [18]: liste_t, liste_y = euler(f, 0, 1, .01, 1000)
```

On peut tracer la courbe de la solution apporchée que l'on peut comparer à la courbe de la solution exacte. En effet, on montre sans peine que l'unique solution de ce problème de Cauchy est la fonction $x \mapsto e^{\sin(x)}$.

```
In [19]: import matplotlib.pyplot as plt

In [20]: from numpy import exp, sin, linspace

In [21]: plt.figure();

# Tracé de la solution approchée
In [22]: plt.plot(liste_t, liste_y, color='red', label='Solution approchée');

# Tracé de la solution exacte
In [23]: x = linspace(0, 10, 1000)

In [24]: y = exp(sin(x))

In [25]: plt.plot(x, y, '--', color='blue', label='Solution exacte');

In [26]: plt.legend();

In [27]: plt.show()
```



Bien entendu, l'approximation affine $y'(t+\Delta t) \approx f(t) + f'(t)\Delta t$ est d'autant meilleur que Δt est petit.

On peut adapter la méthode au cas d'un système différentiel d'ordre 1. Soit par exemple à résoudre le système différentiel suivant.

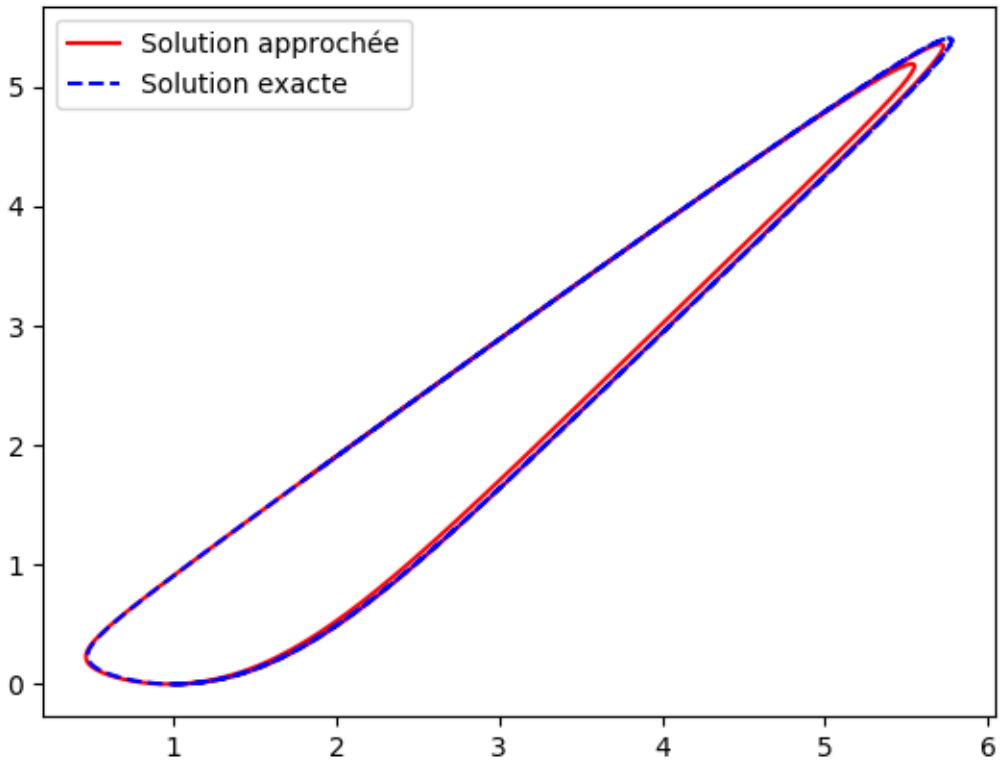
$$\begin{cases} x' = \cos(t)x + \sin(t)y \\ y' = \sin(t)x + \cos(t)y \\ (x(0), y(0)) = (1, 0) \end{cases}$$

```
In [28]: def euler(f, t0, X0, pas, nb):
    ....:     t = t0
    ....:     X = X0
    ....:     liste_t = [t]
    ....:     liste_X = [X]
    ....:     for _ in range(nb):
    ....:         X = [x + u * pas for x, u in zip(X, f(t, X))]
    ....:         t += pas
    ....:         liste_t.append(t)
    ....:         liste_X.append(X)
    ....:     return liste_t, liste_X
    ....:
```

```
In [29]: from math import cos, sin

In [30]: f = lambda t, X: [cos(t) * X[0] + sin(t) * X[1], sin(t) * X[0] + cos(t) *␣
    ↪X[1]]
```

```
In [31]: liste_t, liste_X = euler(f, 0, [1, 0], .01, 1000)
```

```
In [32]: import matplotlib.pyplot as plt

In [33]: from numpy import exp, sin, cos, sinh, cosh

In [34]: plt.figure();

# Tracé de la solution approchée
In [35]: plt.plot(*zip(*liste_X), color='red', label='Solution approchée');

# Tracé de la solution exacte
In [36]: t = linspace(0, 10, 1000)

In [37]: x = exp(sin(t)) * cosh(1 - cos(t))

In [38]: y = exp(sin(t)) * sinh(1 - cos(t))

In [39]: plt.plot(x, y, '--', color='blue', label='Solution exacte');

In [40]: plt.legend();

In [41]: plt.show()
```



On sait qu'il est toujours possible de ramener une équation différentielle scalaire d'ordre strictement supérieur à 1 à un système différentiel d'ordre 1.

Par exemple, si l'on désire résoudre le problème de Cauchy

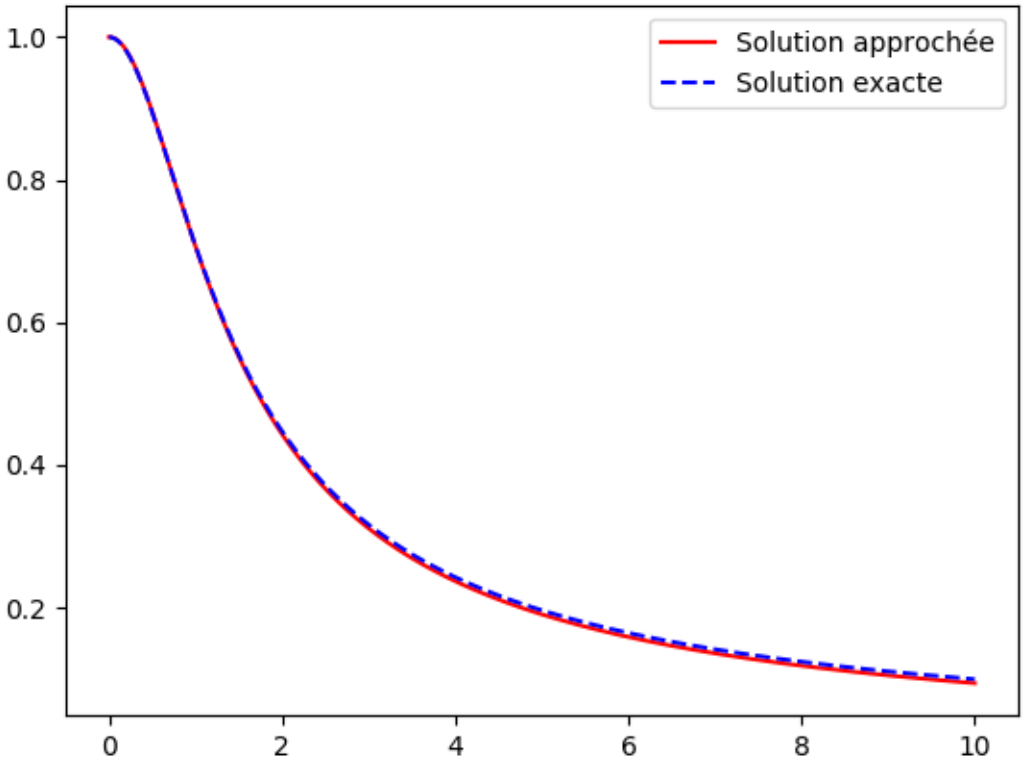
$$\begin{cases} y'' + \dfrac{2t}{1+t^2}y' + \dfrac{1}{(1+t^2)^2}y = 0 \\ (y(0), y'(0)) = (1, 0) \end{cases}$$

on peut se ramener au système différentiel d'ordre 1 suivant

$$\begin{cases} y' = z \\ z' = -\dfrac{2t}{1+t^2}z - \dfrac{1}{(1+t^2)^2}y \\ (y(0), z(0)) = (1, 0) \end{cases}$$

```
In [42]: f = lambda t, X: [X[1], -X[0] / (1 + t**2)**2 - 2 * t / (1 + t**2) *X[1]]

In [43]: liste_t, liste_X = euler(f, 0, [1, 0], .01, 1000)
```

```
In [44]: import matplotlib.pyplot as plt

In [45]: from numpy import sqrt

In [46]: plt.figure();

# Tracé de la solution approchée
In [47]: plt.plot(liste_t, [X[0] for X in liste_X], color='red', label='Solution␣
↪approchée');

# Tracé de la solution exacte
In [48]: t = linspace(0, 10, 1000)

In [49]: y = 1/sqrt(1 + t**2)

In [50]: plt.plot(t, y, '--', color='blue', label='Solution exacte');

In [51]: plt.legend();

In [52]: plt.show()
```



5.3 Arithmétique

5.3.1 Décomposition d'un entier dans une base

L'écriture d'un entier $n \in \mathbb{N}^*$ dans une base b où b est un entier supérieur ou égal à 2 est une écriture de la forme

$$n = \sum_{k=0}^{p} a_k b^k$$

où les a_k sont des entiers compris entre 0 et $b-1$ et où $a_p \neq 0$. Par exemple, la décomposition en base 8 de 4621 est

$$1455 = 7 \times 8^0 + 5 \times 8^1 + 6 \times 8^2 + 2 \times 8^3$$

On peut obtenir la liste $[a_0, a_1, \ldots, a_p]$ à l'aide d'une suite de divisions euclidiennes par b. Par exemple,

$$\begin{aligned} 1455 &= 7 + 8 \times 181 \\ 181 &= 5 + 8 \times 22 \\ 22 &= 6 + 8 \times 2 \end{aligned}$$

donc

$$1455 = 7 + 8 \times (5 + 8 \times (6 + 8 \times 2)) = 7 + 5 \times 8 + 6 \times 8^2 + 2 \times 8^3$$

```
In [1]: def decomp(n, b):
   ...:     l = []
   ...:     while n != 0:
   ...:         l.append(n % b)
   ...:         n //= b
   ...:     return l
   ...:

In [2]: decomp(24189, 10 )
Out[2]: [9, 8, 1, 4, 2]

In [3]: n, b = 3678, 7

In [4]: l = decomp(n, b)

In [5]: l
Out[5]: [3, 0, 5, 3, 1]

# On vérifie que la liste convient effectivement
In [6]: sum(a * b ** i for i, a in enumerate(l))
\\\\\\\\\\\\\\\\\\\\\\\\Out[6]: 3678
```

5.3.2 Calcul de PGCD

On peut implémenter l'algorithme d'Euclide en Python.

```
In [7]: def pgcd(a, b):
   ...:     while b!= 0:
   ...:         a, b = b, a % b
   ...:     return abs(a)    # Le pgcd doit être positif
   ...:

In [8]: pgcd(30, 12)
Out[8]: 6

In [9]: pgcd(30, -12)
\\\\\\\\\\Out[9]: 6
```

De même, on peut implémenter l'algorithme d'Euclide étendu qui, en plus du pgcd, donne des coefficients d'une relation de Bézout, c'est-à-dire des entiers u et v tels que

$$au + bv = \text{pgcd}(a, b)$$

```
In [10]: def bezout(a, b):
    ....:     s, t, u, v = 1, 0, 0, 1
    ....:     while b != 0:
    ....:         q = a // b
    ....:         a, s, t, b, u, v = b, u, v, a - q * b, s - q * u, t - q * v
    ....:     return (a, s, t) if a > 0 else (-a, -s, -t)      # Le pgcd doit être␣
↪positif
    ....:

In [11]: bezout(30, 12)
Out[11]: (6, 1, -2)
```

```
In [12]: bezout(30, -12)
\\\\\\\\\\\\\\\\\\\Out[12]: (6, -1, -3)
```

5.3.3 Exponentiation rapide

Il s'agit ici de calculer efficacement une puissance entière d'un objet mathématique (nombre ou matrice par exemple). Un algorithme naïf serait le suivant.

```
In [13]: def exponentiation(x, n):
   ....:     a = 1
   ....:     for _ in range(n):
   ....:         a *= x
   ....:     return a
   ....:

In [14]: exponentiation(3, 5)
Out[14]: 243
```

Il est clair que cet algorithme nécessite n multiplications pour calculer une puissance $n^{\text{ème}}$.

On peut proposer un léger raffinement pour éviter une multiplication.

```
In [15]: def exponentiation2(x, n):
   ....:     if n == 0:
   ....:         return 1
   ....:     a = x
   ....:     for _ in range(n-1):
   ....:         a *= x
   ....:     return a
   ....:

In [16]: exponentiation2(3, 5)
Out[16]: 243
```

Mais on peut être beaucoup plus efficace. On remarque que toute puissance x^n peut en fait s'écrire comme un produit de puissances de la forme x^{2^k} : en effet, n peut s'écrire comme une somme d'entiers de la forme 2^k.

$$x^{13} = x \times x^{12} = x \times (x^2)^6 = x \times (x^4)^3 = x \times x^4 \times x^8$$

Il suffit alors de calculer successivement les x^{2^k} par des élevations au carré puis de multiplier ces puissances entre elles. Par exemple, dans l'exemple précédent, on utilise :

— 3 multiplications pour calculer successivement x^2, x^4 et x^8 ;
— 2 multiplications pour effectuer le produit de x, x^4 et x^8.

On effectue en tout 5 multiplications au lieu de 12.

Pour obtenir la décomposition de x^n en produit de facteurs de la forme x^{2^k}, il suffit de décomposer n en base 2. On utilise donc un algorithme similaire à l'algorithme de décomposition binaire. En effet, en notant q et r le quotient et le reste de la division euclidienne de n par 2

$$x^n = \begin{cases} (x^2)^q \text{ si } r = 0 \\ x \times (x^2)^q \text{ si } r = 1 \end{cases}$$

```
In [17]: def exponentiation_rapide(x, n):
   ....:     a = 1
```

```
   ....:     p = x
   ....:     while n > 0:
   ....:         if n % 2 == 1:
   ....:             a *= p
   ....:         p *= p
   ....:         n //= 2
   ....:     return a
   ....:

In [18]: exponentiation_rapide(3, 5)
Out[18]: 243
```

On peut également proposer un raffinement pour gagner une multiplication. En effet, à la dernière itération, l'instruction `p *= p` est inutile puisque cette dernière valeur de `p` ne sera pas utilisée. De plus, à la dernière itération, `n` vaut `1` qui est impair donc l'instruction `a *= p` sera obligatoirement effectuée.

```
In [19]: def exponentiation_rapide2(x, n):
   ....:     if n == 0:
   ....:         return 1
   ....:     a = 1
   ....:     p = x
   ....:     while n > 1:
   ....:         if n % 2 == 1:
   ....:             a *= p
   ....:         p *= p
   ....:         n //= 2
   ....:     return a * p
   ....:

In [20]: exponentiation_rapide2(3, 5)
Out[20]: 243
```

5.3.4 Evaluation de polynômes

La méthode naïve pour évaluer un polynôme P en un scalaire x consiste à calculer les différentes puissances de x puis à les multiplier par les coefficients de P correspondant puis à effectuer la somme de ces produits.

Par exemple, pour évaluer $5X^3 + 4X^2 - 3X + 7$ en un scalaire x, on calcule succesivement :
- les puissances de x, à savoir x^2 et x^3 (2 multiplications) ;
- mes produits $-3x$, $4x^2$ et $5x^3$ (3 multiplications) ;
- la somme de 7, $-3x$, $4x^2$ et $5x^3$ (3 additions).

Mais les calculs peuvent être menés plus astucieusement en remarquant que :

$$7 - 3X + 4X^2 + 5X^3 = 7 + X\left(-3 + 4X + 5X^2\right) = 7 + X\left(-3 + X\left(4 + 5X\right)\right)$$

On calcule alors successivement :
- $s_1 = 4 + 5x$ (1 multiplication et 1 addition) ;
- $s_2 = -3 + xs_1$ (1 multiplication et 1 addition) ;
- $s_3 = 7 + xs_2$ (1 multiplication et 1 addition).

On a gagné deux multiplications par rapport à la méthode précédente et on comprend bien que le gain sera d'autant plus grand que le polynôme est de degré élevé.

L'algorithme décrit dans l'exemple précédent s'appelle la **méthode de Hörner**. Pour implémenter cet algorithme, on représente un polynôme par la liste de ses coefficients rangés par ordre décroissant de degré. Par exemple, la liste `[1, 2, 3]` représente le polynôme $X^2 + 2X + 3$.

```
In [21]: def horner(poly, x):
   ....:     s = 0
   ....:     for c in poly:
   ....:         s = s * x + c
   ....:     return s
   ....:

# On évalue le polynôme X²+2X+3 en 4
In [22]: horner([1, 2, 3], 4)
Out[22]: 27
```

Si l'on préfère représenter un polynôme par la liste de ses coeffeficients par ordre de degré *croissant*, on peut toujours utiliser la fonction `reversed` qui fait ce que son nom indique [1].

```
In [23]: def horner(poly, x):
   ....:     s = 0
   ....:     for c in reversed(poly):
   ....:         s = s * x + c
   ....:     return s
   ....:

# On évalue le polynôme 1+2X+3X² en 4
In [24]: horner([1, 2, 3], 4)
Out[24]: 57
```

5.4 Probabilités

5.4.1 Statistiques

Le calcul de la moyenne est on ne peut plus simple : il s'agit de la somme des éléments de la liste divisée par le nombre d'éléments de cette liste [1]. De manière plus formmelle, la moyenne m d'une liste $(x_1, \ldots, x_n)$ de nombres est

$$m = \frac{1}{n} \sum_{k=1}^{n} x_k$$

```
In [1]: def moyenne(liste):
   ...:     somme = 0
   ...:     for el in liste:
   ...:         somme += el
   ...:     return somme / len(liste)
```

On peut également utiliser du slicing : `lst[::-1]` est également la liste `lst` « renversée ».

Evidemment, Python dispose déjà deux fonctions permettant de calculer aisément la moyenne d'une liste de nombres. On peut par exemple utiliser la fonction `sum` qui, comme son nom l'indique, calcule la somme des éléments d'une liste (ou plus généralement d'un objet de type itérable).

```
In [21]: moyenne = lambda liste: sum(liste) / len(liste)

In [22]: moyenne([1, 2, 3])
Out[22]: 2.0
```

Le module `numpy` dispose même d'une fonction `mean` (*moyenne* en anglais).

```
In [23]: from numpy import mean

In [24]: mean([1, 2, 3])
Out[24]: 2.0
```

```
   ...:

In [2]: moyenne([1, 2, 3])
Out[2]: 2.0
```

On peut donner deux expressions de la variance v d'une liste $(x_1, \ldots, x_n)$ de nombres dont on dispose déjà de la moyenne m.

$$v = \left(\frac{1}{n}\sum_{k=1}^{n} x_k^2\right) - m^2 = \frac{1}{n}\sum_{k=1}^{n}(x_k - m)^2$$

En utilisant la première expression, on peut par exemple donner cette fonction de calcul de la variance [2].

```
In [3]: def variance(liste):
   ...:     s1, s2 = 0, 0
   ...:     n = len(liste)
   ...:     for el in liste:
   ...:         s1 += el
   ...:         s2 += el * el
   ...:     return s2 / n - (s1 / n) ** 2
   ...:

In [4]: variance([1, 2, 3])
Out[4]: 0.666666666666667
```

On peut également utiliser une des fonctions de calcul de moyenne définies précédemment.

```
In [5]: variance = lambda liste: moyenne([el ** 2 for el in liste]) - moyenne(liste)␣
↪** 2

In [6]: variance([1, 2, 3])
Out[6]: 0.666666666666667
```

Si l'on préfère, on peut également utiliser la deuxième expression de la variance.

```
In [7]: def variance(liste):
   ...:     m = moyenne(liste)
   ...:     return moyenne([(el - m) ** 2 for el in liste])
   ...:

In [8]: variance([1, 2, 3])
Out[8]: 0.6666666666666666
```

5.4.2 Simuler une variable aléatoire

Dans la suite, on fera appel à la fonction `random` du module `random` qui renvoie un flottant tiré aléatoirement dans l'intervalle $[0, 1[$.

Bien entendu, le module `numpy` dispose déjà d'une fonction ad hoc : la fonction `var`.

```
In [25]: from numpy import var

In [26]: var([1, 2, 3])
Out[26]: 0.6666666666666666
```

```
In [9]: from random import random

In [10]: [random() for _ in range(10)]
Out[10]:
[0.5981969750479801,
 0.5981658454201241,
 0.20412949464258667,
 0.4103391200893315,
 0.05595213540993549,
 0.17857162606843002,
 0.11801955693516719,
 0.08763524371094311,
 0.369772361763542,
 0.4686147822739093]
```

Cela nous permettra de simuler des variables aléatoires connaissant leurs lois [3].

On cherche dans un premier temps à simuler une variable aléatoire X **à valeurs dans un ensemble fini**, disons $\{0, \dots, n-1\}$ où $n \in \mathbb{N}^*$, dont on connaît la loi, c'est-à-dire les valeurs de $\mathbb{P}(X = k)$ pour $k \in \{0, \dots, n-1\}$.

On construit pour cela une fonction prenant pour argument la loi d'une telle variable aléatoire sous la forme d'une liste de réels positifs de somme 1.

```
In [11]: def simul(loi):
    ....:     proba = random()
    ....:     s = 0
    ....:     for i, p in enumerate(loi):
    ....:         s += p
    ....:         if proba < s:
    ....:             return i
    ....:
```

```
In [12]: [simul([.3, .5, .2]) for _ in range(20)]
Out[12]: [1, 1, 0, 0, 2, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 2, 1, 1, 2, 2]
```

On désire maintenant simuler une variable aléatoire X **à valeurs dans un ensemble dénombrable**, disons $\mathbb{N}$, dont on connaît la loi, c'est-à-dire les valeurs de $\mathbb{P}(X = k)$ pour $k \in \mathbb{N}$.

La loi de cette variable aléatoire ne peut alors plus être représentée sous la forme d'une liste finie ; on la représente donc comme une fonction d'argument un entier n et renvoyant $\mathbb{P}(X = n)$.

```
In [13]: def simul(loi):
    ....:     proba = random()
    ....:     s = loi(0)
    ....:     n = 0
    ....:     while proba >= s:
    ....:         n += 1
    ....:         s += loi(n)
    ....:     return n
    ....:
```

```
In [14]: from math import factorial, exp

# Simulation d'une loi de Poisson
In [15]: poisson = lambda l: lambda n: exp(-l) * l**n / factorial(n)
```

A nouveau, le module `numpy.random` dispose déjà de fonctions permettant de simuler la plupart des lois classiques.

```
In [16]: [simul(poisson(2)) for _ in range(20)]
Out[16]: [0, 0, 1, 4, 1, 2, 4, 3, 2, 3, 2, 1, 1, 1, 3, 2, 1, 1, 2, 1]
```

Pour terminer, on peut facilement simuler une variable suivant une **loi binomiale** puisque l'on sait qu'elle est de même loi qu'une somme de variables de Bernoulli indépendantes.

```
In [17]: def bernoulli(p):
   ....:     return 1 if random() < p else 0
   ....:

In [18]: def binomiale(n, p):
   ....:     return sum(bernoulli(p) for _ in range(n))
   ....:

In [19]: [binomiale(5, .8) for _ in range(20)]
Out[19]: [4, 5, 3, 4, 4, 5, 3, 4, 4, 4, 4, 3, 5, 4, 4, 5, 3, 5, 4, 5]

In [20]: [binomiale(5, .2) for _ in range(20)]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]: [1, 3,␣
↪1, 2, 2, 1, 1, 1, 1, 2, 3, 2, 0, 3, 0, 0, 1, 1, 0, 1]
```

5.5 Matrices

Dans ce paragraphe, les matrices seront représentées par des listes de listes. Par exemple, la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ sera représentée par la liste de listes `[[1, 2, 3], [4, 5, 6]]` [1].

5.5.1 Produit matriciel

```
In [1]: def produit(A, B):
   ...:     return [[sum(L[k] * B[k][j] for k in range(len(L))) for j in␣
↪range(len(B[0]))] for L in A]
   ...:

In [2]: A = [[1, 2, 3], [4, 5, 6]]

In [3]: B = [[1, 2], [3, 4], [5,6]]

In [4]: produit(A, B)
Out[4]: [[22, 28], [49, 64]]

In [5]: produit(B, A)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]: [[9, 12, 15], [19, 26, 33], [29, 40, 51]]
```

5.5.2 Opérations élémentaires

On définit plusieurs opérations élémentaires sur les lignes d'une matrice.

— l'échange de lignes $L_i \leftrightarrow L_j$

[1] Le module `numpy` possède un type `matrix` permettant de simplifier grandement les fonctions suivantes. Il possède d'ailleurs également un sous module `numpy.linalg` regroupant de nombreuses fonctions ayant trait à l'algèbre linéaire sur les matrices.

```
In [6]: def echange_lignes(M, i, j):
   ...:     M[i], M[j] = M[j], M[i]
   ...:     return M
   ...:
```

— la transvection $L_i \leftarrow L_i + \lambda L_j$

```
In [7]: def transvection_ligne(M, i, j, l):
   ...:     M[i] = [M[i][k] + l * M[j][k] for k in range(len(M[i]))]
   ...:     return M
   ...:
```

— la dilatation $L_i \leftarrow \lambda L_i$

```
In [8]: def dilatation_ligne(M, i, l):
   ...:     M[i] = [coeff * l for coeff in M[i]]
   ...:     return M
   ...:
```

Avertissement : Les fonctions précédentes, modifient la matrice donnée en argument puisqu'une liste est un objet mutable.

5.5.3 Algorithme du pivot de Gauss

A l'aide des opérations élémentaires précédemment définies, on peut alors définir une fonction appliquant l'algorithme du pivot de Gauss à une matrice pour la mettre sous forme *échelonnée*.

Pour des raisons de stabilité numérique, on recherche le pivot de valeur absolue maximale.

```
In [9]: def recherche_pivot_lignes(M, i):
   ...:     m = abs(M[i][i])
   ...:     j = i
   ...:     for k in range(i + 1, len(M)):
   ...:         if abs(M[i][j]) > m:
   ...:             j = k
   ...:     return j
   ...:
```

```
In [10]: def pivot_lignes(M):
   ....:     for i in range(len(M)):
   ....:         j = recherche_pivot_lignes(M, i)
   ....:         if j != i:
   ....:             echange_lignes(M, i, j)
   ....:         if M[i][i] != 0:
   ....:             for j in range(i + 1, len(M)):
   ....:                 transvection_ligne(M, j, i, -M[j][i] / M[i][i])
   ....:     return M
   ....:
```

Note : Le test `if M[i][i] != 0:`, s'il est correct en théorie, est en fait ridicule en pratique. Puisque l'on ne travaille qu'avec des valeurs approchées, un pivot nul en théorie (si l'on effectuait des calculs exacts) ne sera jamais nul en pratique.

```
In [11]: M = [[1, 2, 3, 4], [5, 6, 7, 8], [6, 8, 10, 12], [4, 4, 4, 4]]

In [12]: pivot_lignes(M)
Out[12]:
[[1, 2, 3, 4],
 [0.0, -4.0, -8.0, -12.0],
 [0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0]]
```

Note : On pourrait alors utiliser la forme échelonnée pour calculer le rang d'une matrice : il suffirait alors de compter le nombre de lignes non nulles. Mais à nouveau, il n'est pas évident de savoir en pratique si une ligne est réellement nulle puisqu'on a accès qu'à des valeurs approchées de ses coefficients.

5.5.4 Résolution de systèmes linéaires

On considère un **système de Cramer** sous forme matricielle $AX = B$ où A est une matrice inversible, B une matrice colonne donnée et X une matrice colonne inconnue. Pour résoudre ce système, il suffit dans un premier temps de mettre la matrice $\big(A \mid B\big)$ sous forme échelonnée. On peut utiliser la fonction `pivot_lignes` précédemment définie mais on aura également besoin d'une fonction permettant de concaténer une matrice carrée (sous forme d'une liste de listes) et une matrice colonne (sous forme d'une liste).

```
In [13]: def concatenation_vecteur(A, B):
   ....:     return [A[i] + [B[i]] for i in range(len(A))]
   ....:
```

Une fois que le pivot de Gauss a été effectué sur la matrice $\big(A \mid B\big)$, il faut effectuer un pivot « à rebours » pour déterminer la solution du système $AX = B$.

```
In [14]: def pivot_lignes_rebours(M):
   ....:     for i in reversed(range(len(M))):
   ....:         dilatation_ligne(M, i, 1 / M[i][i])
   ....:         for j in range(i):
   ....:             transvection_ligne(M, j, i, -M[j][i])
   ....:     return M
   ....:
```

La matrice colonne solution est alors la dernière colonne de la matrice obtenue, qu'il faut donc extraire.

```
In [15]: def extract_vecteur(M):
   ....:     return [L[-1] for L in M]
   ....:
```

On peut alors définir une fonction d'arguments une matrice inversible A et une matrice colonne B renvoyant l'unique solution du système $AX = B$.

```
In [16]: def resolution(A, B):
   ....:     M = concatenation_vecteur(A, B)
   ....:     pivot_lignes(M)
   ....:     pivot_lignes_rebours(M)
   ....:     return extract_vecteur(M)
   ....:
```

```
In [17]: A = [[1, -1, 2], [3, 2, 1], [2, -3, -2]]

In [18]: B = [5, 10, -10]

In [19]: resolution(A, B)
Out[19]: [1.0, 2.0, 3.0]
```

5.5.5 Inversion d'une matrice

On peut également utiliser l'algorithme du pivot de Gauss pour inverser une matrice : on transforme une matrice inversible en la matrice identité en effectuant l'algorithme du pivot de Gauss puis l'algorithme du pivot de Gauss « à rebours ». On récpercute les opérations effectuées sur une matrice identité de même taille que A, qui est alors transformée en l'inverse de la matrice initiale. Pour effectuer aissément les mêmes opérations sur les lignes d'une matrice A et la matrice identité I, on forme la matrice $\left(A \mid I\right)$.

```
In [20]: def concat_identite(A):
   ....:     return [A[i] + [1 if j== i else 0 for j in range(len(A))] for i in␣
↪range(len(A))]
   ....:
```

Après les pivots, il reste à extraire la matrice inverse.

```
In [21]: def extract_inverse(M):
   ....:     return [L[len(M):] for L in M]
   ....:
```

On peut alors proposer la fonction suivante.

```
In [22]: def inverse(A):
   ....:     M = concat_identite(A)
   ....:     pivot_lignes(M)
   ....:     pivot_lignes_rebours(M)
   ....:     return extract_inverse(M)
   ....:
```

```
In [23]: A = [[1, 5, 6], [2, 11, 19], [3, 19, 47]]

In [24]: B = inverse(A)

In [25]: B
Out[25]: [[156.0, -121.0, 29.0], [-37.0, 29.0, -7.0], [5.0, -4.0, 1.0]]

In [26]: produit(A, B)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]: [[1.
↪0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]

In [27]: produit(B, A)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
```

5.5.6 Calcul du déterminant

On peut également se servir du pivot de Gauss pour calculer le déterminant d'une matrice carrée. En effet, le déterminant est invariant par transvection et échange de lignes et le déterminant d'une matrice triangulaire est le produit de

ses coefficients diagonaux [2].

```
In [28]: def determinant(M):
   ....:     pivot_lignes(M)
   ....:     p = 1
   ....:     for i in range(len(M)):
   ....:         p *= M[i][i]
   ....:     return p
   ....:
```

```
In [29]: M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

In [30]: determinant(M)
Out[30]: -0.0
```

5.6 Tris

5.6.1 Tri par insertion

```
In [1]: def tri_insertion(tab):
   ...:     for i in range(1,len(tab)):
   ...:         val = tab[i]
   ...:         pos = i
   ...:         while pos > 0 and tab[pos - 1] > val:
   ...:             tab[pos] = tab[pos-1]
   ...:             pos -= 1
   ...:         tab[pos] = val
   ...:
```

```
In [2]: from numpy.random import randint

In [3]: tab = randint(100, size=20)

In [4]: tab
Out[4]:
array([18, 62, 15, 54, 39, 40, 94, 28, 42, 79, 12, 56, 85, 32, 94, 77, 13,
        9, 15, 97])

In [5]: tri_insertion(tab)

In [6]: tab
Out[6]:
array([ 9, 12, 13, 15, 15, 18, 28, 32, 39, 40, 42, 54, 56, 62, 77, 79, 85,
       94, 94, 97])
```

5.6.2 Tri rapide

```
In [7]: def tri_rapide(tab):
   ...:     if tab == []:
   ...:         return []
```

[2] On pourrait penser à calculer le déterminant via la formule qui l'exprime en fonction des coefficients de la matrice ou à l'aide d'un développement par rapport à une ligne ou une colonne mais on verra dans le chapitre ??? que c'est nettemment moins efficace que le pivot de Gauss

```
   ...:     else:
   ...:         pivot = tab[0]
   ...:         t1, t2 = [], []
   ...:         for x in tab[1:]:
   ...:             if x < pivot:
   ...:                 t1.append(x)
   ...:             else:
   ...:                 t2.append(x)
   ...:         return tri_rapide(t1) + [pivot] + tri_rapide(t2)
   ...:
```

```
In [8]: from numpy.random import randint

In [9]: tab = randint(100, size=20)

In [10]: tab
Out[10]:
array([12, 69, 31, 90, 14, 60, 50, 56, 61,  0, 87, 30, 36, 42, 89, 80, 67,
       15, 19, 56])

In [11]: tri_rapide(tab)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[0, 12, 14, 15, 19, 30, 31, 36, 42, 50, 56, 56, 60, 61, 67, 69, 80, 87, 89, 90]
```

5.6.3 Tri par fusion

```
In [12]: def tri_fusion(tab):
   ....:     if len(tab) < 2:
   ....:         return tab
   ....:     else:
   ....:         m = len(tab)//2
   ....:         return fusion(tri_fusion(tab[:m]), tri_fusion(tab[m:]))
   ....:
```

```
In [13]: def fusion(t1, t2):
   ....:     i1, i2, n1, n2 = 0, 0, len(t1), len(t2)
   ....:     t=[]
   ....:     while i1 < n1 and i2 < n2:
   ....:         if t1[i1] < t2[i2]:
   ....:             t.append(t1[i1])
   ....:             i1 += 1
   ....:         else:
   ....:             t.append(t2[i2])
   ....:             i2 += 1
   ....:     if i1 == n1:
   ....:         t.extend(t2[i2:])
   ....:     else:
   ....:         t.extend(t1[i1:])
   ....:     return t
   ....:
```

```
In [14]: from numpy.random import randint

In [15]: tab = randint(100, size=20)
```

```
In [16]: tab
Out[16]:
array([58, 48,  6, 96,  0, 44, 28,  1, 49, 90, 19, 46, 78, 52, 70,  0, 62,
        8, 69, 98])

In [17]: tri_fusion(tab)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[0, 0, 1, 6, 8, 19, 28, 44, 46, 48, 49, 52, 58, 62, 69, 70, 78, 90, 96, 98]
```

CHAPITRE 6

Un peu de théorie

6.1 Preuve d'un algorithme

Prouver un algorithme signifie démontrer qu'un algorithme effectue bien la tâche qui lui est demandée.

6.1.1 Terminaison

Il s'agit de montrer que l'algorithme se termine bien. La question se pose notamment lorsque l'algorithme comporte une boucle conditionnelle `while`. La plupart du temps, cela revient à montrer qu'une quantité entière positive décroît strictement à chaque itération de la boucle (une suite d'entiers naturels strictement décroissante est nécessairement *finie*). Cette quantité est généralement appelé le **variant**.

On peut par exemple étudier l'algorithme d'Euclide pour le calcul du pgcd.

```
def pgcd(a, b):
  while b!= 0:
      a, b = b, a % b
  return a
```

On suppose que l'argument `b` est un entier naturel. En notant b_k la valeur de `b` à la fin de la $k^{\text{ème}}$ itération (b_0 désigne la valeur de `b` avant d'entrer dans la boucle), on a $0 \leq b_{k+1} < b_k$ si $b_k > 0$. La suite (b_k) est donc une suite strictement décroissante d'entiers naturels : elle est finie et la boucle se termine.

6.1.2 Correction

Il s'agit de savoir si l'algorithme fournit bien la réponse attendue. Si l'algorithme comporte une boucle, on recherche généralement une quantité ou une propriété qui ne change pas au cours des itérations : on parle d'**invariant de boucle**.

A nouveau, on peut étudier l'algorithme d'Euclide.

```
def pgcd(a, b):
  while b!= 0:
      a, b = b, a % b
  return a
```

On note a_k et b_k les valeurs de `a` et `b` à la fin de la $k^{\text{ème}}$ itération (a_0 et b_0 désignent les valeurs de `a` et `b` avant d'entrer dans la boucle). Or, si $a = bq + r$, il est clair que tout diviseur commun de a et b est un diviseur commun de b et r et réciproquement. Notamment, $a \wedge b = b \wedge r$. Ceci prouve que $a_k \wedge b_k = a_{k+1} \wedge b_{k+1}$. La quantité $a_k \wedge b_k$ est donc bien un invariant de boucle. En particulier, à la fin de la dernière itération (numérotée N), $b_N = 0$ de sorte que $a_0 \wedge b_0 = a_N \wedge b_N = a_N \wedge 0 = a_N$. La fonction `pgcd` renvoie donc bien le pgcd de `a` et `b`.

6.2 Complexité

CHAPITRE 7

Index et tables

— genindex
— modindex
— search