Programmation en python, ou comment réussir sans trop dépendre des autres

Gabriel Genest
Jérémie Gince

Destiné aux étudiants en sciences voulant apprendre ou perfectionner leurs aptitudes de programmation.

Version 0.1.2021

Table des matières

1 Préface

1.1 Bonjour

Bienvenue dans le fabuleux monde universitaire, où les professeurs ont tendance à tenir beaucoup de choses pour acquises, où les jours passent trop rapidement et où vos meilleurs amis risquent de vous demander des réponses de devoir à minuit la veille de la remise. Malgré cela, vous vous engagez dans un voyage fantastique et ferez des rencontres inoubliables, que ce soit avec vos pairs et même avec vos professeurs ! Comme les sciences contemporaines font de plus en plus appel au monde numérique, il n'est donc pas surprenant de voir un cours d'introduction à la programmation dans le cursus des apprentis scientifiques.

Mais est-ce que ce cours est suffisant pour être à l'aise et autonome ? Nous pensons que non. La structure actuelle des cours donnés est trop générale et de base. Les cours sont surtout centrés sur les bases de la programmation et les travaux donnés sont intéressants et amusants à compléter, mais ne préparent pas à faire des graphiques, ni des intégrales numériques. De plus, au cours des dernières années, c'est avec un certain regret que nous avons constaté que plusieurs étudiants ne savaient pas effectuer de simples opérations mathématiques ou afficher des graphiques. Il n'est pas rare non plus de voir d'autres étudiants finir le cours d'introduction à la programmation et dire «J'ai fait le cours de python et je ne sais même pas ce qu'est une classe», alors que la programmation orientée objet est probablement le paradigme le plus utilisé (et le plus utile) de nos jours dans le domaine du développement informatique et que c'est essentiellement une partie du cours (voire le cours au complet) qui n'est pas comprise. Il existe bien sûr des cours plus poussés, comme des cours de *C++* ou le cours *physique numérique* (pour ceux étant en physique, comme les auteurs), mais les premiers ne sont pas extrêmement pertinents dans le parcours typique du scientifique ordinaire, alors que le second ne met pas nécessairement l'emphase sur la programmation, mais plus sur les méthodes numériques (de plus, il n'est pas offert au gens extérieurs à physique/génie physique). C'est pourquoi nous vous préparons ce petit guide qui se veut plus qu'un simple *cheat sheet*, mais moins qu'un manuel complet comme celui qu'on vous recommande dans certains cours de programmation (qui peut être inutile pour certains, mais essentiel pour d'autres).

Veuillez garder en tête que nous ne sommes aucunement des experts en la matière, seulement des passionnés de programmation. Ceci dit, il se peut donc que certaines erreurs se soient glissées durant l'écriture ou que nous ne soyons pas à jour sur certaines possibilités du langage *Python*. Sur ce, merci de lire cet ouvrage et nous espérons sincèrement qu'il vous sera utile. N'hésitez pas à nous contacter si vous avez des questions ou des commentaires.

De plus, étant donné qu'il s'agit (en ce moment) d'une version encore en développement, il se peut fort probablement que beaucoup de fautes de frappe soient présentes. Nous en sommes désolés.

1.2 Utilité de la programmation

La programmation joue un rôle prépondérant dans la société actuelle. Presque tous les objets de nos quotidiens font appel à une quelconque programmation. On pense bien sûr aux ordinateurs et téléphones, mais même une télévision, une voiture et un ventilateur sont des objets *programmés*. La programmation ne fait pas tout le temps référence à écrire du code, on peut aussi parler d'objets ayant certaines fonctionnalités. C'est en quelque sorte grâce à la programmation qu'on peut appuyer sur un simple bouton pour activer la rotation d'un ventilateur ou activer une alarme incendie. La programmation est partout.

En sciences, comme dans bien d'autres domaines, on utilise la programmation pour simplifier des tâches souvent trop répétitives, comme compter jusqu'à cent et faire une simple manipulation mathématique, ou trop complexe, comme résoudre les équations de Lorenz. C'est pourquoi avoir des bases solides en programmation est essentiel si on veut réussir en sciences au XXI^{e} siècle. Il faut aussi comprendre que la programmation aide aussi sur papier. La logique de programmation est assez universelle. Elle est basée sur la logique mathématique. Donc, passer du crayon au papier peut s'avérer plus facile qu'on le pense, même si au premier regard cela semble perdu d'avance. De plus, apprendre à programmer, c'est apprendre à penser à différentes manières de résoudre un problème, à *déboguer* ses problèmes, à effectuer des recherches efficaces et à obtenir des solutions à ses problèmes. Souvent, ce ne sont pas les solutions attendues, mais rien n'est plus satisfaisant que de construire son propre programme, régler les problèmes qu'il cause et finalement obtenir ce qu'on désire. Somme toute, la programmation c'est l'indépendance, la compréhension et le développement de soi-même.

1.3 Pourquoi Python ?

Python est possiblement le langage de programmation le plus utilisé dans les domaines académiques et scientifiques. Il est facile d'approche pour les débutants et pas trop simpliste pour les professionnels. Il permet le meilleur de la programmation, comme la programmation orientée objet, l'héritage (qui peut être multiple), le développement d'interface graphique, etc. Malheureusement, dans certaines conditions, il pourrait être pertinent d'utiliser le polymorphisme, mais *Python* ne le permet pas. Malgré cela, le langage offre des avantages quand on le compare à d'autres langages :

Premièrement, on n'a pas besoin de gérer la compilation du code, contrairement à *C/C++* où il faut s'assurer d'avoir un bon compilateur, ce qui implique une compilation du code avant de l'exécuter. Avec *Python*, il suffit d'avoir un interpréteur (voir section **??**).

Deuxièmement, *Python* s'occupe automatiquement de la gestion de la mémoire avec ce qu'on appelle le *garbage collector*, ou le collecteur d'ordures en français. On retrouve ce genre de concept dans d'autres langages très utilisés comme *Java* ou *C#*, mais pas en *C/C++*. Cela est très intéressant, car lorsqu'on doit gérer la mémoire et qu'on n'est pas très à l'aise avec la programmation, cela peut devenir un cauchemar et causer des fuites de mémoire.

Troisièmement, le typage en *Python* est dynamique. Comme vous l'avez vu (ou allez voir) dans le cours d'introduction à la programmation, il existe différents types de données de base et vous n'avez presque pas à vous en soucier, car *Python* «décide» pour vous lequel utiliser. Dans la plupart des langages utilisés, comme *C*, *C++*, *C#*, *Java*, le programmeur doit spécifier quel est le type d'une variable.

Finalement, *Python* est gratuit à utiliser (dans tes dents *MATLAB*) et est *open source*, c'est-à-dire que le code source des librairies externes est accessible, contrairement, encore une fois, aux autres langages mentionnés plus haut.

2 Assignation et affectation

Cette section sert principalement à montrer comment assigner une valeur quelconque à une variable. Elle se veut courte et de base, donc un lecteur à l'aise avec cela peut passer à une section suivante. On peut voir l'assignation et l'affectation comme la fonction mathématique `y = ax+b`, on «donne» la valeur `ax+b` à la variable `y`. En suivant cette logique, on peut écrire :

```
a = 34
b = "toto"
c = 12/8
```

En *Python*, il est facile d'assigner des valeurs à des variables, car on n'a pas à se soucier des types. Par exemple, on n'a pas à spécifier si on veut que la variable contienne un entier, une chaîne de caractères, etc. L'opérateur = sers à «donner» une valeur à une variable. On peut par la suite accéder à cette valeur en faisant appel à la variable et en effectuant certaines opérations qui dépendent du type de la variable. On verra plus loin quels sont les principaux types de variables et qu'est-ce qu'on peut faire avec. On peut aussi jongler entre différents types dans une variable. Contrairement à d'autres langages, où le type d'une variable ne peut changer au cours de sa vie, on peut écrire sans problème :

```
a = "toto"
a = 2e-12
a = 4j + 3.4
a = 4 < 2
```

Les types de variables sont parfois (voire souvent, mais tout dépend du contexte) peu importants, car *Python* est assez «intelligent» pour déterminer lui-même de quel type il s'agit, notamment par l'entremise du *duck typing*[1] et le typage dynamique. Par contre, l'utilisation de librairies telles que *NumPy*[2] ou *Pandas* peut rendre le type très important. Par exemple, utiliser certaines fonctions de *Scikit-Image*, pour le traitement d'image, fonctionnent mieux ou sont seulement définies pour des structures de données d'entiers signés sur 8 bits.

Dans certains cas plus complexes, assigner le contenu d'une variable à une autre variable, puis modifier l'originale peut aussi modifier la nouvelle. Par exemple :

```
a = [1,2,3]
b = a
a.append(4) # on ajoute 4 a la liste a
print(a)
print(b)
```

1. Technique basée sur le principe «Si ça marche comme un canard et fait "quack", ça doit être un canard». Donc, *Python* se base sur «est-ce que l'objet courant possède une certaine méthode (ou plusieurs méthodes)» lorsqu'on veut faire des manipulations, alors que d'autres langages vérifient le type de l'objet, et non son contenu.

2. *NumPy* ajoute même de nouveaux types que le langage de base ne possède pas. Par exemple, le `int` de base (qui est non signé) est encodé sur 64 bits (donc on peut dire int64), alors que *NumPy* ajoute des entiers non signés (int8, int16, ...), des entiers signés (uint8, uin16, ...) et plus encore.

Le résultat est :

```
[1,2,3,4]
[1,2,3,4]
```

Dans ce cas (et dans plusieurs autres), *Python* n'a même pas effectué de copie, `a` et `b` sont donc deux variables faisant référence aux mêmes adresses mémoires.

Si on change a (b), on modifie du même coup b (a). Ce qui amène à parler de copie de surface et de copie profonde. Ce sujet est un peu plus poussé, donc un lecteur débutant peut passer cette sous-section.

2.1 Copie de surface vs copie profonde

Nous ne rentrerons pas vraiment dans les détails du pourquoi du comment, mais l'essentiel sera présent.

2.1.1 Copie de surface

La plus simple des deux types, la copie de surface ne fait que créer un nouvel objet composé et insère dans ce nouvel objet les références vers les objets contenus dans la copie originale. Par exemple, si on a un objet liste et qu'on veuille en faire une copie de surface, on se crée une nouvelle liste et on y insère les références de ce qui est contenu dans la liste originale. Si l'objet composé original contient d'autres objets composés (par exemple, une liste de listes), la copie de surface ne sera pas suffisante. Il faut faire une copie profonde. Par contre, si l'objet contient des objets simples (nombres, chaînes de caractères, etc.), la copie de surface est bonne.

2.1.2 Copie profonde

La copie profonde est plus complexe que la copie de surface. En effet, non seulement on effectue une copie de surface pour le conteneur (l'objet contenant les objets), mais on en fait aussi pour le contenu (pour les objets internes). On y arrive récursivement, donc si on a une liste de listes, la liste contenant les listes est copiée, mais les listes internes aussi. Si on a une liste de listes de listes, on copie la liste qui contient les listes de listes, puis les listes contenant les listes et les listes...Bref, tout est copié. Il ne devrait plus y avoir de problèmes lorsqu'on modifie un objet composé.

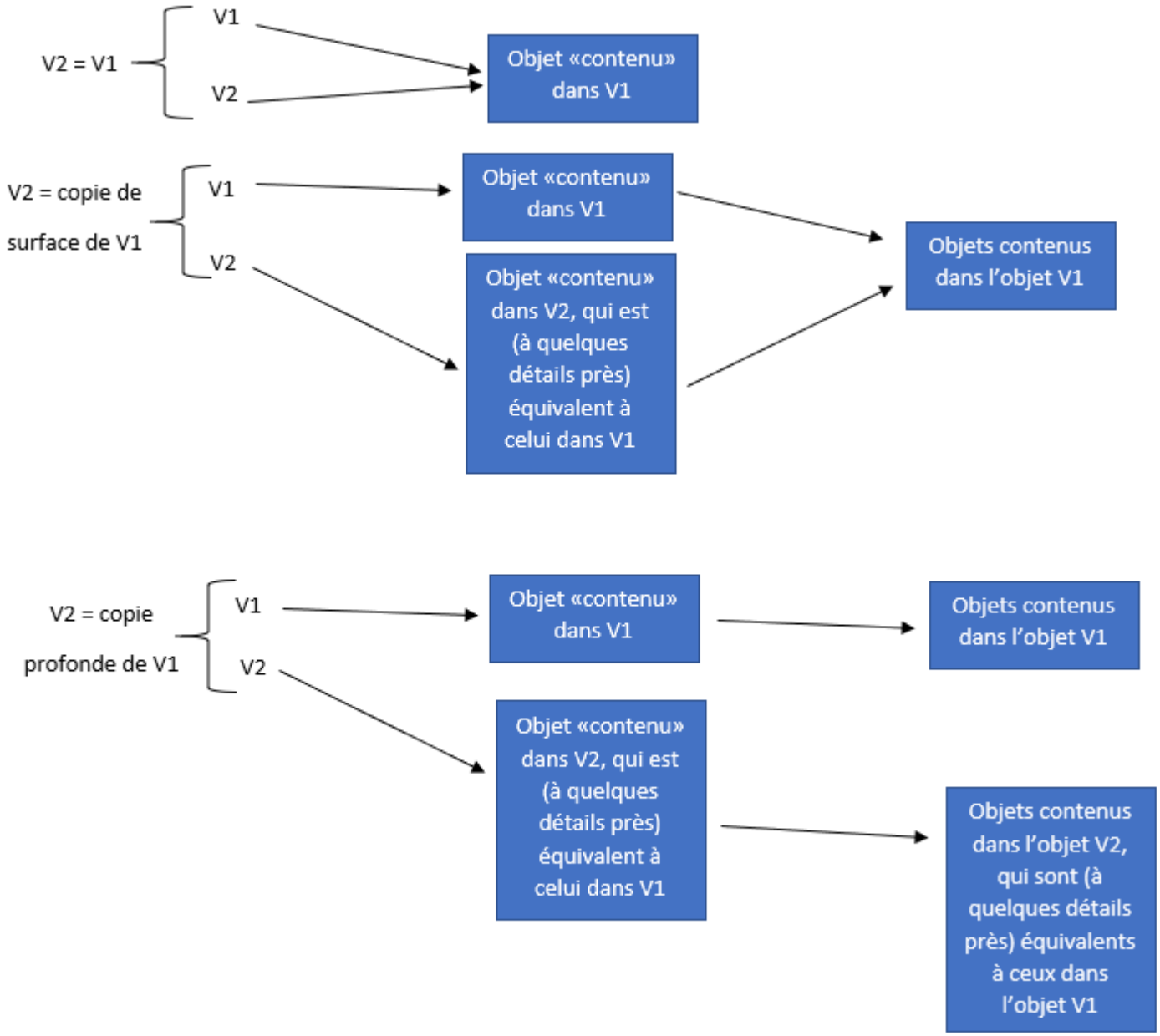


FIGURE 2.1: Différences entre l'assignation, la copie de surface et la copie profonde

3 Objets built-in

En *Python*, même si le typage est dynamique, il faut quand même qu'il y ait des types. Bien que certaines librairies ajoutent d'autres types (comme *Numpy*), *Python* permet les entiers (positifs et négatifs, le type **int**), les nombres réels (positif et négatif, le type **float**) et les chaînes de caractères (le type `string`, ou plus précisément à *Python*, **str**). Il y a aussi un autre type de base qui est extrêmement important, c'est le booléen. Ce type est nécessaire dans l'évaluation de comparaisons et d'égalités, car c'est le booléen qui contient les valeurs *Vrai* (`True`) et *Faux* (`False`). Ces types sont à la base de la programmation en *Python*. Contrairement à d'autres langages, ces types ne sont pas vraiment *de base*, mais cela dépasse les objectifs de ce guide. On peut aussi parler du type **complex**. Comme on s'en doute, ce type représente les nombres complexes. On peut les utiliser de la manière suivante :

```
nombre_complexe = complex(3, 4)
```

Il existe aussi un caractère spécial, `j`, qui, lorsque derrière un entier ou un nombre fractionnaire, représente l'unité imaginaire. Outre passer par **complex**`(3, 4)`, on peut écrire `3 + 4j` et on obtient le même nombre complexe.

3.1 Utilisation des nombres de bases

Il existe quelques opérations arithmétiques de base qu'on peut effectuer avec les nombres (**int**, **float**, **complex**). Voici un court résumé de ce qui peut être fait :

- Addition :
  - **int** + **int**, ça donne un **int**
  - **int** + **float** (et l'inverse), ça donne un **float**
  - **float** + **float**, ça donne un **float**
  - **complex** + **float** (et l'inverse), ça donne un **complex**
  - **complex** + **int** (et l'inverse), ça donne un **complex**
  - **complex** + **complex**, ça donne un **complex**
- Soustraction :
  - **int** - **int**, ça donne un **int**
  - **int** - **float** (et l'inverse), ça donne un **float**
  - **float** - **float**, ça donne un **float**
  - **complex** - **float** (et l'inverse), ça donne un **complex**
  - **complex** - **int** (et l'inverse), ça donne un **complex**
  - **complex** - **complex**, ça donne un **complex**
- Multiplication :
  - **int** * **int**, ça donne un **int**
  - **int** * **float** (et l'inverse), ça donne un **float**
  - **float** * **float**, ça donne un **float**

  - **complex** * **float** (et l'inverse), ça donne un **complex**
  - **complex** * **int** (et l'inverse), ça donne un **complex**
  - **complex** * **complex**, ça donne un **complex**
- Division :
  - **int** / **int**, ça donne un **float**
  - **int** / **float** (et l'inverse), ça donne un **float**
  - **float** / **float**, ça donne un **float**
  - **complex** / **float** (et l'inverse), ça donne un **complex**
  - **complex** / **int** (et l'inverse), ça donne un **complex**
  - **complex** / **complex**, ça donne un **complex**
- Division entière (division qui donne la partie entière inférieure ou égale) [3] :
  - **int** // **int**, ça donne un **int**
  - **int** // **float** (et l'inverse), ça donne un **float**
  - **float** // **float**, ça donne un **float**
  - Pour les **complex**, la division entière n'est pas définie
- Exposant :
  - **int** ** **int**, ça donne un **int**
  - **int** ** **float** (et l'inverse), ça donne un **float**
  - **float** ** **float**, ça donne un **float**
  - **complex** ** **float** (et l'inverse), ça donne un **complex**
  - **complex** ** **int** (et l'inverse), ça donne un **complex**
  - **complex** ** **complex**, ça donne un **complex**
- Modulo [4] (donne le reste de la division euclidienne) :
  - **int** % **int**, ça donne un **int**
  - **int** % **float** (et l'inverse), ça donne un **float**
  - **float** % **float**, ça donne un **float**
  - **complex** % **float** (et l'inverse), ce n'est pas défini
  - **complex** % **int** (et l'inverse), ce n'est pas défini
  - **complex** % **complex**, ce n'est pas défini

Il est aussi important de savoir qu'il est possible en *Python* d'écrire rapidement des nombres en notation scientifique, grâce au caractère `e`, lorsqu'il est précédé par un **int** ou un **float** et aussi suivi par un **int** ou un **float**. Par exemple, `2.25e-122` est équivalent à 2.25×10^{-12} ou encore `12E89` est équivalent à 12×10^{89}.

Mélanger ces opérations arithmétiques de bases avec des objets plus complexes peut provoquer des comportements indésirables. Par exemple, effectuer `[1,2,3] * 3` $\neq$ `[3,6,9]` ! Plus de précisions seront données lorsqu'il sera question de ces objets.

3. Même si la division entière peut donner un nombre de type **float**, il est important de se souvenir que ce **float** est un entier (même s'il y a une virgule et une décimale, décimale qui est en fait 0 !)

4. Même si le modulo peut donner un nombre de type **float**, il est important de se souvenir que ce **float** est un entier (même s'il y a une virgule et une décimale, décimale qui est en fait 0 !)

3.2 Les strings

Le `string` est un type de base de python. C'est un conteneur séquentiel de `char`. Mais qu'est-ce qu'un `char` ? C'est tout simplement un caractère textuel que l'on connaît tous ou bien un `string` de longueur 1 (en python). Voici les valeurs possibles que peut prendre un `char` :

- Toutes les lettres de l'alphabet minuscules et majuscules.
- Tous les chiffres de 0 à 9
- \\ : le caractère \
- \' : le caractère '
- \" : le caractère "
- \a : un bip (son)
- \b : le caractère d'effacement
- \f : le caractère de saut de page
- \n : le caractère de saut de ligne
- \r : le caractère de retour au début de la ligne
- \t : le caractère de tabulation horizontale
- \v : le caractère de tabulation verticale
- \xhh : le caractère de valeur hh codée en notation hexadécimale
- \ooo : le caractère de valeur ooo codée en notation octale
- Tout caractère UTF-8

On peut construit un `char` de la façon suivante :

```
my_char = 'a'
```

ou bien comme suit :

```
my_char = chr(97)  # prendra la valeur 'a'
```

et on peut le visualiser comme en utilisant la commande `print(my_char)`. Il est vraiment intéressant de pouvoir écrire des caractères et de pouvoir les manipuler, mais il serait encore plus intéressant de pouvoir les concaténer (les mettre un à la suite de l'autre) afin de pouvoir manipuler un ensemble de `char` dans le but de travailler avec des phrases ou des textes complets. C'est ce qu'offrent les `string` (ou `str` dans le langage *Python*).

On peut imager un `string` comme suit : Le conteneur place les caractères de façon séquentielle et

M	a		S	t	r	i	n	g

on peut le construire de façon très similaire au `char` : `my_string = "Ma String"`. Maintenant, comment manipule-t-on ces chaînes de caractères ? C'est très simple, il suffit d'utiliser des méthodes pré implémenté dans python.

Voici quelques exemples :

On peut les concaténer.

```
s1 = "string1"
s2 = "string2"
s1p2 = s1 + s2
print(s1p2)
# out: string1string2
```

On peut accéder à leur longueur.

```
s = "string"
s_lenght = len(s)
print(s_lenght)
# out: 6
```

On peut les convertir en d'autres types.

```
s1 = "72"
s_int = int(s1)
print(f"{type(s_int)}, {s_int}")
# out: <class 'int'>, 72
s2 = "72.42"
s_float = float(s2)
print(f"{type(s_float)}, {s_float}")
# out: <class 'float'>, 72.42
```

On peut accéder a un certain caractère.

```
s = "string"
c = s[0]
print(c)
# out: s
```

On peut accéder à des sous-chaînes.

```
s = "string"
ss1 = s[1:5]
print(ss1)
# out: trin
```

On utilise les crochets de cette façon :

- [i] : un seul élément situé à l'indice i.
- [début :fin] : tous les éléments depuis l'indice début jusqu'à l'indice fin-1.
- [début :fin :k] : idem précédent, mais en sautant de k indice(s) à chaque élément.

On peut formater les chaînes en utilisant la méthode **format**.

```
y = 3.141592
print('---{0:8.2f}---{0:^8.4f}---{0:8.2e}---'.format(y))
# out: ---    3.14---  3.1416  ---3.14e+00---
s = "string"
sf = f"my {s}"
print(sf)
# out: my string
s = "{}: {} --> {}".format("hey", "quoi?", "dah!")
print(s)
# out: hey: quoi? --> dah!
```

On peut aussi utiliser d'autres méthodes associées aux `string`.

- Chercher le nombre d'occurrences d'une sous-chaîne avec `count`
- Chercher l'indice de début d'une sous-chaîne avec `find`
- Déterminer si tous les caractères sont des chiffres `isdigit`
- Convertir en minuscules avec `lower`
- Convertir en majuscules avec `upper`
- Remplacer toutes les occurrences d'une sous-chaîne avec `replace`
- Convertir en une liste de mots avec `split`

Finalement, il est extrêmement important de savoir que les objets **`str`** sont immuables, c'est-à-dire qu'on ne peut changer les caractères à l'intérieur. Par exemple, il est impossible de faire :

```
s = "Hello World"
s[0] = "e"
```

Il y aura une erreur. De plus, comme on peut accéder à une sous-chaîne à l'aide des crochets, les **`str`** permettent l'indexation, donc on peut itérer sur ces objets, accéder à chaque caractère, un après l'autre (voir la section 5 sur les boucles).

Il est aussi très important de savoir que les opérateurs (+, ∗, etc.) n'ont pas le même comportement avec les **`str`** que ceux qu'ils ont avec les nombres. Les principaux opérateurs importants (et pas mal les seuls) qu'on peut utiliser avec les chaînes de caractères sont + et ∗. Le premier sert à concaténer des **`str`**, alors que le second sert à répéter une chaîne un certain nombre de fois :

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)
print(s1 * 2 + " " + s2 * 2)
```

```
Hello World
HelloHello WorldWorld
```

3.3 Les listes

Les **list** sont des conteneurs séquentiels qui s'utilisent essentiellement comme les **string**. La différence est que les **list** peuvent contenir n'importe quel objet et non seulement des **char**. De plus, les listes sont mutables, donc on peut en changer le contenu. Le fonctionnent des listes est alors très simple puisqu'on peut essentiellement accéder aux éléments dans une liste de la même façon qu'un **string**. Il est cependent interessant de savoir comment itérer dans une liste sachant qu'on peut faire de même pour une **string**.
Nous pouvons itérer dans une liste de la façon basique, c'est-à-dire en utilisant l'index des éléments.

```
element_2 = int()
element_3 = float()
# declaration d'une liste: var = [e_0, e_1, e_2, ..., e_n]
# ou bien var = list() qui appelle le constructeur par defaut
my_list = [list(), "element_1", element_2, element_3]

for i in range(len(my_list)):
    print(f"{i}, {my_list[i]}")
```

```
0, []
1, element_1
2, 0
3, 0.0
```

Ou bien nous pouvons itérer sans nous soucier de l'index et de la longueur de la liste.

```
element_2 = int()
element_3 = float()
my_list = [list(), "element_1", element_2, element_3]

for e in my_list:
    print(f"{my_list.index(e)}, {e}")
```

```
0, []
1, element_1
2, 0
3, 0.0
```

Ou nous pouvons itérer en utilisant la méthode génératrice "**enumerate**" qui génère un tuple contenant en premier élément, l'index et en deuxième, l'item courant de la liste.

```
element_2 = int()
element_3 = float()
my_list = [list(), "element_1", element_2, element_3]

for i, e in enumerate(my_list):
    print(f"{i}, {e}")
```

```
0, []
1, element_1
2, 0
3, 0.0
```

En ce qui concerne les opérateurs, encore une fois on peut utiliser + et ∗. Le premier sert une fois de plus à concaténer deux listes, alors que le second sert à répéter un certain nombre de fois les éléments d'une liste :

```
l1 = [1, 2, 3]
l2 = [-1, -2, -3]
print(l1 + l2]
print(l1 * 2)
```

```
[1, 2, 3, -1, -2, -3]
[1, 2, 3, 1, 2, 3]
```

Il faut toutefois faire attention à l'opérateur ∗ lorsqu'on utilise une liste d'objets composés, comme une liste de listes. Par exemple, si on veut créer un tableau en deux dimensions dont les éléments sont les entiers entre −2 et 2, inclusivement, on peut écrire :

```
tableau = [[-2, -1, 0, 1, 2]] * 5
```

Par contre, si on veut modifier la première ligne pour remplacer le −2 par 1000, on modifie toutes les lignes du tableau :

```
tableau[0][0] = 1000
print(tableau)
```

```
[[1000, -1, 0, 1, 2], [1000, -1, 0, 1, 2],
    [1000, -1, 0, 1, 2], [1000, -1, 0, 1, 2], [1000, -1, 0, 1, 2]]
```

On remarque donc que toutes les lignes sont modifiées. Il faut faire attention.

3.4 Les set

Le **set** est une collection d'items uniques, non ordonnés et non indexés. Un **set** se comporte alors exactement comme un ensemble mathématique. Voici un exemple de manipulation des ensembles.

```
if __name__ == '__main__':
    A: set = {0, 1, 5, 6, 8}  # Creation du set A
    B: set = {2, 5, 8, 7, 5, 9}  # Creation du set B

    print(A, B)  # Affichage des set A et B

    # Acceder aux item dans le set A
    for x in A:
        print(x)

    # Ajout d'un item au set A
    A.add(11)
    print(A)
    A.add(6)
    print(A)

    # ajout de plusieurs items au set B
    B.update([2, 9, 11, 10, 25, 33])
    print(B)

    # On enleve un item du set B
    B.discard(33)
    print(B)
```

```
{0, 1, 5, 6, 8} {2, 5, 7, 8, 9}
0
1
5
6
8
{0, 1, 5, 6, 8, 11}
{0, 1, 5, 6, 8, 11}
{33, 2, 5, 7, 8, 9, 10, 11, 25}
{2, 5, 7, 8, 9, 10, 11, 25}
```

Méthodes	Description
add(elem)	Ajout de l'élément "elem" à l'ensemble.
clear()	Retire tous les éléments de l'ensemble de façon à avoir l'ensemble vide.
difference(other : set)	Retourne un **set** contenant seulement la différence entre le **set** courant et le **set** "other". Ce qui veut dire que le **set** retourné contiendra seulement les éléments existants dans le **set** courant et non dans le **set** "other".
difference_update(other : set)	Retire les éléments du **set** courant existants dans les deux **set**.
discard(elem)	Retire l'élément "elem" du **set** courant.
intersection(set1, set2, ... etc)	Retourne un **set** conetenant les éléments existants dans tous les **set** en paramètre ainsi que le **set** courant.
intersection_update(set1, set2, ... etc)	Retire tous les éléments qui ne sont présents dans les **set** en paramètre ainsi que le **set** courant.
isdisjoint(other : set)	Retourne une valeur booléenne vraie si les deux **set** sont disjoint et faux sinon.
issubset(other : set)	Retourne une valeur booléenne vraie si le **set** courant est un sous-ensemble du **set** en paramètre et faux sinon.
issuperset(other : set)	Retourne une valeur booléenne vraie si le **set** en paramètre est un sous-ensemble du **set** courant et faux sinon.
pop()	Retire un élément aléatoire du **set** courant et le retourne.
remove(elem)	Retire l'élément "elem" du **set** courant et soulève un erreur si l'élément n'est pas présent dans le **set** courant.
symmetric_difference(other : set)	Retourne un **set** contenant tous les éléments des deux **set** sauf ceux présents dans les deux exactement.
symmetric_difference_update(other : set)	Retire les éléments qui existent dans les deux **set** et ajoute les autres élément au **set** courant.
union(set1, set2, ...)	Ajoute les éléments de tous les **set** en paramètres.
update(other : set)	Ajoute les éléments du **set** en paramètre.

Tableau 3.1: Méthodes de l'objet **set**

3.5 Les dictionnaires

Les dictionnaires sont des objets excessivement utiles en python. En effet, ils permettent de contenir une série de couples (clé, valeur) qui peuvent être facilement accessibles et changeable. Ce conteneur sert à contenir des valeurs associées à une clé unique, ce qui veut dire que l'ensemble des clés du conteneur forme l'équivalent d'un **set**. De plus, il faut savoir que le dictionnaire est non ordonné, donc il n'y a pas d'index associés aux éléments internes. Seulement de montrer comment déclarer un dictionnaire va nous aider à visualiser comment fonctionne cet objet.

```
my_dict = {
"annee": 1997,
"animaux": ["chat", "chien", "oiseaux", "autruche", "chameaux"],
"cost": 9.99,
"un_tuple": (12, 99,),
5 : "5",
7: "sept"
}
```

On peut donc voir que l'on peut insérer plusieurs types d'objets en tant que clé ou de valeur. On aurait aussi pu mettre un dictionnaire comme valeur à une certaine clé. Voici un exemple d'utilisation d'un dictionnaire.

```
NameToAge: dict = {
    "Philipe": 12,
    "Alex": 33,
    "Leonidas": 35,
    "Achilles": 26
}

minimumSalary: float = 12.35
NameToSalary: dict = {name: minimumSalary for name in NameToAge.keys()}

Augmentation_ratio: float = 0.1

NameToSalary["Leonidas"] = NameToSalary.get("Leonidas") \
                           + NameToSalary.get("Leonidas") \
                           * Augmentation_ratio

print(NameToSalary)
```

```
{'Philipe': 12.35, 'Alex': 12.35,
'Leonidas': 13.584999999999999, 'Achilles': 12.35}
```

Maintenant, afin de manipuler de façon plus efficace les dictionnaires, voici les diverses méthodes de cet objet.

Méthodes	Description
clear()	Retire tous les éléments du dictionnaire courant.
copy()	Retourne une copy du dictionnaire courant.
dict.fromkeys(keys, value)	Méthode statique retournant un dictionnaire avec l'itérable "keys" et assigne la valeur "value" pour chacune des clés.
get(keyname, value=None)	Retourne la valeur associée à la clé "keyname" du dictionnaire courant et si celle-ci n'est pas présente, la méthode retourne une certaine valeur par défaut "value".
items()	Retourne une liste contenant tous les pairs (clé, valeur) du dictionnaire courant.
keys()	Retourne un "view object" contenant tous les clés du dictionnaire courant. Ce "view object" est un objet se traitant comme une liste permettant de voir les clés du dictionnaire courant en tout temps, alors du moment que celui-ci change, le "view object" change aussi.
pop(keyname, defaultvalue)	Retire l'élément à la clé "keyname" et le retourne. Si celui-ci n'existe pas la valeur "defaultvalue" est retourné.
popitem()	Retire un tuple aléatoire (clé, valeur) du dictionnaire courant et le retourne.
setdefault(keyname, value=None)	Retourne la valeur associée à la clé "keyname" du dictionnaire courant et si celle-ci n'est pas présente, la méthode retourne une certaine valeur par défaut "value".
update(iterable)	Ajoute au dictionnaire courant les pairs (clé, valeur) de l'iterable.
values()	Retourne un "view object" des clés du dictionnaire courant. Ce "view object" est un objet se traitant comme une liste permettant de voir les clés du dictionnaire courant en tout temps, alors du moment que celui-ci change, le "view object" change aussi.

Tableau 3.2: Méthodes du l'objet **dict**

3.6 Les tuples

Un tuple est un conteneur ordonné et non changeable. C'est-à-dire qu'un tuple possède une indexation de chacun de ses éléments comme les **list** et qu'une fois qu'un tuple est créé, il est impossible de changer les items à l'intérieur. Voici quelques exemples d'utilisation :

Création et indexation.

```
    my_tuple: tuple = ("kevin", "Alphonse", "Benjamain", "Moutarde")

    print(my_tuple[0])
    print(my_tuple[-1])

out:
    kevin
    Moutarde
```

Savoir si un objet est dans un certain tuple.

```
    print("Moutarde" in my_tuple)
    print("Ketchup" in my_tuple)

out:
    True
    False
```

Création d'un tuple à un seul item.

```
    One_item_in_parentheses = (17.3)
    One_item_tuple = (17.3, )  # Notice the comma ','

    print(type(One_item_in_parentheses))
    print(type(One_item_tuple))

out:
    <class 'float'>
    <class 'tuple'>
```

Concaténation de deux tuples.

```
    Concat_tuple = my_tuple + One_item_tuple
    print(Concat_tuple)

out:
    ('kevin', 'Alphonse', 'Benjamain', 'Moutarde', 17.3)
```

Convertir une liste en tuple.

```
    simple_list: list = [1, 2, 3, 4, 5]
    simple_tuple: tuple = tuple(simple_list)
    print(type(simple_tuple))

out:
    <class 'tuple'>
```

Méthodes	Description
count(value)	Retourne le nombre de fois que l'objet "value" se retrouve dans le tuple courant.
index(value)	Retourne l'index du premier objet "value".

TABLEAU 3.3: Méthodes de l'objet **tuple**

3.7 Exercices sur les objets built-ins

3.7.1 Types de bases

Il peut y avoir plusieurs réponses possibles !

— Que donnent les opérations suivantes ? Quel est le type ?

```
a = 12
b = 12.0
a + b # = ?
```

```
a = 1/10
b = 1e2
a + b # = ?
```

```
a = 4j
b = 4.5
b % a # = ?
```

— Quelle est la valeur manquante ainsi que son type ?

```
a = # ?
b = 12.3
a + b # = 6j
```

```
a = # ?
b = 3
a // b # = 1.0
```

```
a = # ?
b = 2
a % b == 0 # = True
```

3.7.2 Les strings

Il peut y avoir plusieurs valeurs possibles !

— Qu'affichent les opérations suivantes ?

```
print("Hello" + " " + "World")
```

```
a = "BONJOUR"
print(a.lower())
```

```
a = "AHAHAHAH!"
print(a.count("A"))
```

```
a = "1 2 3 4 5 6"
for char in a.split(" "):
    print(char)
```

```
a = " hello world "
b = " hello world "
print(a.replace(" ", "") == b.strip())
```

```
a = "change le premier caractere par w"
a[0] = "w"
print(a)
```

```
print("q" in "abcdeftghijklmnoprstuvwxyz")
```

3.7.3 Les listes

Il peut y avoir plusieurs réponses valides !

— Quels sont les éléments de la liste ainsi que leur type ?

```
a = 123
b = "toto"
c = "est malade"
d = 4j + 33
liste = [[a + 12, a], b + c, str(d)]
```

— Que se passe-t-il avec les expressions suivantes ? (Expliquer ce qui se passe. Par exemple, l'élément i passe de la valeur x à y). Quelle(s) est (sont) la (les) nouvelle(s) liste(s) ? (S'il y eut des changements)

```
liste = [1, 2, 3, 4, 5]
liste[1:2] = [2, 3]
```

```
liste = ["1", 1, 2, 3, 4, 5]
del liste[0]
```

```
liste = [1, 2, 3, 4]
liste2 = liste
liste[0] = -1
liste2[0] = 1
```

```
liste = [1, 2, 3, 4]
liste2 = liste.copy()
liste[-1] = -1000
liste2[-1] = 1e34
```

```
liste = [1, [1, 2, 3, 4], 3, 4]
liste2 = liste.copy()
liste[0] = -1
liste[1][:] = "\t"
```

```
liste = [ i for i in range(100)]
del liste[:]
```

```
liste = [1,2,3,4,5]
liste2 = liste.copy()
del liste
print(liste2)
```

```
liste = [1,2,3]
liste2 = liste
del liste2
print(liste)
```

```
liste = [1,2,3]
liste2 = liste
del liste
print(liste2)
```

3.7.4 Les autres objets built-ins

Il peut y avoir plusieurs réponses valides !

— À partir des deux listes suivantes (une de clés et une de valeurs), construisez un dictionnaire (le premier élément de la liste de clés correspond à la clé du premier élément de la liste de valeurs).

```
cles = [1, 2, 3, 4, 5, 6, 235, 3467654, -90, 9]
valeurs = ["a", "b", "c", "d", "DDDDD!", "ASDFGCIXOKMA", "dfgtr", "asdf",
           (9,9), ["Hello", "World"]]
```

Ce traitement peut se faire en une seule ligne avec la fonction **zip**

— Quel est le résultat des affichages suivants ?

```
s = {1, 2, 3, 4, 5, 6, 7, 8, 9}
print(s)
```

```
s2 = {1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1, 2}
print(s2)
```

```
s3 = {"a", "b"}
print(s3.pop())
print(s3)
```

4 Les conditions

Les conditions sont des éléments à la base de la programmation. C'est grâce à elles qu'on peut avoir des algorithmes complexes et qu'on peut avoir des intelligences artificielles.

4.1 Les opérateurs de comparaison

En *Python*, comme dans tout langage qui se respecte, il existe une certaine quantité non négligeable d'opérateurs de comparaison. Voici un tableau listant les principaux, ainsi qu'une brève description de ce qu'il font :

`a < b`	Vérifie si a est plus petit que b
`a > b`	Vérifie si a est plus grand que b
`a <= b`	Vérifie si a est plus petit ou égal à b
`a >= b`	Vérifie si a est plus grand ou égal à b
`a == b`	Vérifie si a est égal à b
`a != b`	Vérifie si a est différent de b

Il peut être aussi pertinent de parler de l'opérateur **`in`**. On l'utilise avec la syntaxe suivante :

```
a in b
```

où a est n'importe quoi, alors que b doit être un objet itérable, donc qu'on puisse parcourir, comme une liste, un dictionnaire ou une chaîne de caractères. Cet opérateur vérifie si b contient a. Il existe aussi l'opérateur **`is`**, qui est parfois mélangé avec `==`, mais les deux sont très différent. Le premier sert à évaluer l'égalité d'objets, donc si deux objets sont les mêmes, s'ils ont le même **`id`**[5]. On l'utilise principalement pour vérifier si une variable est `None`[6] :

```
a is None
```

Il est important de noter que chaque comparaison est en fait un «calcul», donc elle retourne une valeur, un booléen qui vaut `True` (vrai) ou `False` (faux).

4.2 Non, ou, et

Comme en logique avec les opérateurs $\neg$ (non), $\wedge$ (et) et $\vee$ (ou), il existe des mots réservés qui permettent la combinaison de plusieurs opérateurs de comparaison : **`not`** pour rendre `True`→ `False` (et l'inverse aussi), **`or`** pour joindre deux opérations par le «ou» et **`and`** pour joindre deux opérations par le «et». Voici quelques exemples :

```
a = 10
b = 11
```

5. Le concept derrière **`is`** n'est pas essentiel à comprendre. De plus, il est plus compliqué que simplement vérifier l'id.

6. On pourrait aussi utiliser `==`, mais il est préférable d'utiliser **`is`**.

```
c = a + b
print(a < c or a == c - b)
print(a != b and c >= a)
print(not a == a ** 2)
```

Le résultat est :

```
True
True
True
```

4.3 La condition *si*, et comment imbriquer plusieurs cas

À la base des instructions conditionnelles se trouve **if** (si). La syntaxe à utiliser est :

```
if <condition>:
    <code a executer>
```

La condition peut être relativement simple, comme simplement évaluer si $a < b$ ou beaucoup plus complexe. Bien sûr, si la condition se révèle fausse, le code n'est pas exécuté. Voici un exemple :

```
def estPair(nombre):
    pair = False
    if nombre % 2 == 0:
        pair = True
    return pair
```

On peut aussi complexifier *la patente* :

```
def estBissextile(annee):
    bissextile = False
    if (annee % 4 == 0 and annee % 100 != 0) or annee % 400 == 0:
        bissextile = True
    return bissextile
```

On peut ajouter plusieurs conditions différentes, les emboîter, avec **elif** (sinon si) et **else** (sinon). La logique à prendre est la suivante :

```
if <condition 1>:
    <executer le code si condition 1 respectee>
elif <condition 2>:
    <executer le code si condition 2 respectee>
...
else:
    <code a executer si aucune condition respectee>
```

Il est important de bien saisir les points suivants :

— Si on entre dans le code à exécuter du **if** ou d'un **elif**, on passe par-dessus tous les autres **elif** et le dernier **else**.

— Il ne faut pas avoir plus d'un **else** dans un bloc conditionnel. Un bloc conditionnel débute par un **if** et n'en contient qu'un seul. Si deux **if** se suivent, il s'agit de deux blocs.

Voyons un exemple de ce que ces points impliquent :

```
def unSeulBlocConditionnel(nombre):
    if nombre < 10:
        print("plus petit que 10")
    elif nombre < 20:
        print("plus petit que 20")
    else:
        print("plus grand ou egal a 20")

def deuxBlocsConditionnels(nombre):
    if nombre in range(-100, 100):
        print("intervalle de -100 a 100")
    else:
        print("pas dans interval de -100 a 100")

    if nombre % 2 == 0:
        print("nombre pair")
    else:
        print("nombre impair")
```

Si on fait :

```
print(unSeulBlocConditionnel(9))
print(deuxBlocsConditionnels(9))
```

on a la sortie :

```
plus petit que 10
intervalle de -100 a 100
nombre impair
```

4.4 Les opérations *bitwise*, ou bit-à-bit

Ces opérations sont plus rares. Elles s'utilisent souvent dans des contextes extrêmement spécifiques. On les appelle bit-à-bit, car elles s'opèrent directement sur les bits d'encodage des objets. Par exemple, 7 est équivalent à 111 en binaire ($1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 7$), donc en appliquant un opérateur bit-à-bit sur 7, on se retrouve à manipuler 111 (ou presque, tout ça dépend du nombre de bits d'encodage. Sur 8 bits, on aurait plutôt 00000111). Voici les opérateurs, ce qu'ils font ainsi qu'un exemple :

— `&` : opérateur «et» binaire. Il compare chaque bit entre deux valeurs selon la logique du «et» et donne un résultat. Par exemple :

```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
c = a & b # 12 = 0000 1100
```

— | : opérateur «ou» binaire. Il compare chaque bit entre deux valeurs selon la logique du «ou» et donne un résultat. Par exemple :

```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
c = a | b # 61 = 0011 1101
```

— ^ : opérateur «xor» (ou exclusif) binaire. Il compare chaque bit entre deux valeurs selon la logique du «xor» et donne un résultat. Par exemple :

```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
c = a ^ b # 49 = 0011 0001
```

— ~ : opérateur «non» unaire. Il inverse chaque bit ($0 \to 1$ et $1 \to 0$). Par exemple :

```
a = 60 # 60 = 0011 1100
c = ~a # -61 = 1100 0011
```

— << : opérateur de décalage binaire vers la gauche. Il décale chaque bit d'une certaine valeur. Par exemple :

```
a = 60 # 60 = 0011 1100
c = a << 2 # 240 = 1111 0000
```

— >> : opérateur de décalage binaire vers la droite. Il décale chaque bit d'une certaine valeur. Par exemple :

```
a = 60 # 60 = 0011 1100
c = a >> 2 # 15 = 0000 1111
```

4.5 Exercices

4.5.1 Comparaisons

— Quel est le résultat des comparaisons suivantes ?

```
"a" > "B"
```

```
"a" == "A"
```

```
a = 12
del a
a is None
```

— Quel est le problème dans les expressions suivantes ?

```
a = 12
b = a // 2.66 + 8
a != b # = True
```

```
s1 = "HELLO WORLD"
s2 = " hello world ".replace(" ", "")
s1 == s2 # = True
```

```
"a" in "AAAAAAAAAAAAAAAaAAAAAAAAAAAAAAAAAA" # = False
```

```
valeurs = [i for i in range(100)]
for i in valeurs:
    i in valeurs # = False pour toutes les valeurs
```

4.5.2 Non ou conditions et ?

— Écrivez un bout de code qui vérifie si un nombre quelconque `a` est divisible entièrement par un second nombre `b`.

— Écrivez un bout de code qui prend une chaîne de caractères et qui détermine si elle est une sous-chaîne d'une seconde chaîne de caractères. Par exemple, si on donne `"hello"` comme sous-chaîne et `hello world` comme chaîne, on devrait avoir `True`, car `"hello"` est une sous-chaîne de `"hello world"`

— Écrivez une fonction qui prend en paramètre un nombre

5 Les boucles

En programmation, la notion de boucle est primordiale. Cela permet d'effectuer un certain calcul ou quelconque manipulation tant qu'une condition est respectée. En *Python*, il existe deux sortes de boucles.[7], la boucle **for** et la boucle **while**. Les deux se ressemblent, d'où la possible confusion entre les deux lorsqu'on commence à programmer. Les deux types de boucles sont parfois interchangeables, mais bien souvent on les utilise à des sauces totalement différentes.

5.1 Boucle for

La boucle **for** est utilisée lorsqu'on itère dans un ensemble quelconque. Par exemple, si on veut afficher dix fois *Hello world*, on peut écrire :

```
for _ in range(10):
    print("Hello world")
```

On peut aussi être intéressé à afficher chaque lettre dans un certain mot :

```
mot = "patate"
for lettre in mot:
    print(lettre)
```

On pourrait conclure que la boucle **for** est à prioriser lorsqu'on veut faire un certain bout de code un nombre défini de fois ou lorsqu'on itère sur un ensemble de lettre, de chiffre, d'objets, etc.

5.2 Boucle while

Dans le cas de la boucle **while**, on peut bien sûr effectuer le même type d'itérations que pour la boucle **for**, mais cela est plus long à écrire :

```
iterteur = 0

while iterateur < 10:
    print("Hello world")
    iterateur += 1
```

On l'utilise lorsque la condition d'arrêt est plus complexe que "tant que x plus petit que y" ou que "tant que x est dans y". Par exemple, on pourrait être intéressé à arrêter la boucle seulement lorsqu'un utilisateur entre un caractère valide :

```
caractere_valide = False

while not caractere_valide: # Ceci vaut vrai, donc on entre dans la boucle
    car = input("Entrez la lettre A, B, C, ou D")
```

7. Dans certains autres langages, il en existe trois, la troisième étant `do...while...`. Cette structure, ressemblant vraiment au simple while, a la particularité de toujours s'effectuer une fois, car le bloc d'instructions est exécuté avant la vérification de la condition. Très utile dans certains cas.

```
    if car == "A" or car == "B" or car == "C" or car == "D":
        caractere_valide = True # La condition dans le while est False

print("Sorti de la boucle!")
```

On peut conclure que la boucle **while** pourrait remplacer la boucle **for**, mais elle permet aussi de faire une boucle avec une condition d'arrêt qui peut être très longue et complexe.

5.3 Break et continue

Les mots **break** et **continue** sont des mots réservés du langage, c'est-à-dire qu'ils ne peuvent servir de noms de variables ou de fonctions. Il sont utilisés dans les boucles, que ce soit **for** ou **while**. Le premier est utilisé pour arrêter une boucle avant que sa condition d'arrêt ne soit atteinte, alors que le second est utilisé pour continuer une boucle sans exécuter une certaine partie de code. Voici un exemple d'utilisation :

```
for i in range(1000):
    if i % 2 == 0:
        continue # On ne fait pas le code qui se trouve apres (ici, print)
    print("Nombre impair") # Si i est pair, pas execute

while True: # Boucle possiblement infinie...
    car = input("Entrez la lettre A, B, C, ou D")
    if car == "A" or car == "B" or car == "C" or car == "D":
        break # On quitte la boucle
```

Comme vous pouvez le constater, cette pratique peut parfois alléger le code, mais certaines normes de programmation déconseillent d'utiliser ces mots. De plus, il est possible de faire en sorte qu'une boucle contenant un **break** ou **continue** s'arrête convenablement en posant une condition d'arrêt adéquate.

6 Les fonctions

Les fonctions en programmation sont semblables à ce qu'on connaît d'une fonction mathématique. On peut par exemple écrire une fonction qui permet de calculer $y = mx + b$. Par contre, il faut comprendre quelque chose : *Python* de base ne peut pas effectuer de calcul symbolique, donc lorsqu'on veut calculer y selon $mx + b$, il faut fournir des valeurs de x, on ne peut pas simplement calculer y pour toute valeur de x possible sur l'ensemble R. Pour déclarer une fonction, il faut utiliser le mot réservé **def**, suivi du nom de la fonction. Dans certains cas, il faut spécifier certaines valeurs à la fonction, comme dans le cas de $y = mx + b$ où il faudrait fournir une valeur de m et de b. Pour ce faire, on utilise ce qu'on appelle des paramètres. Voici la syntaxe :

```
def ma_fonction(param1, param2, param3, ...):
    <code a executer>
```

Si on l'applique au cas $y = mx + b$, on aurait :

```
def fonction_lineaire(m, b, x_limite_inf, x_limite_sup, x_pas):
    y = []
    for x in range(x_limite_inf, x_limite_sup, x_pas):
        y.append(m * x + b)
    return y
```

Dans le cas précédent, on remarque la présence de 5 paramètres. On peut mettre autant de paramètres, comme 100 ou bien 0. Par contre, dans le cas à 100 paramètres, il faut faire attention, car trop de paramètres peuvent facilement mêler les autres utilisateurs. Il faut aussi savoir que lorsqu'on ne met aucun paramètre, il faut tout de même mettre des parenthèses après le nom de la fonction. De plus, dans l'exemple précédent, on demande à l'utilisateur de fournir de l'information par rapport au domaine de x. Comme mentionné plus haut, *Python* ne peut pas calculer y pour toutes les valeurs de x, il faut fournir ces valeurs. On procède par l'entremise de la fonction **range**, qui prend au maximum trois arguments : le point de départ, le point d'arrêt (exclu) et le pas (par bons de 1, 2, 1000, ...). Par la suite, on calcul la valeur de y associée à chaque valeur de x, qu'on met dans une liste. Finalement, la dernière ligne sert à retourner un quelconque objet lorsque le code dans la fonction à fini d'être exécuté. Le mot **return** sert seulement à ça. Lorsque la fonction atteint un **return**, on quitte la fonction. Ainsi, si l'on avait la fonction suivante :

```
def quitter_avant_fin():
    a = 10
    if a < 10.000001:
        return "fini"
    b = a ** 4 / (11.1111*1e-5) + 5j
    liste = []
    for i in range(1000000):
        liste.append(i)
    return liste
```

et qu'on exécute le code suivant :

```
print(quitter_avant_fin())
```

on aurait la sortie :

```
fini
```

car le code a rencontré un **return** dans le premier **if**.

Comme vous pouvez le voir, on peut faire pratiquement n'importe quoi dans une fonction. On peut même appeler d'autres fonctions ou modifier des variables externes à la fonction. On peut faire des fonctions sans paramètres ou sans **return**, des fonctions qui s'appellent elles-mêmes (voir 6.2). Il existe déjà plein de fonctions écrites par d'autres utilisateurs ou bien des fonctions *builtins*. La plupart des opérations mathématiques auxquelles vous pouvez penser sont probablement déjà codées dans un module quelconque. Si vous voulez faire du traitement d'image, il existe plein de fonctions. Si vous voulez faire du calcul matriciel, il en existe aussi.

6.1 Yield, ou comment ne pas prendre trop de mémoire

Le mot-clé **yield** est un genre de **return**, dans le sens qu'il sert à «retourner» quelque chose dans une fonction. Par contre, la différence vient dans la manière dont il rend les valeurs ! En effet, **yield** est très différent de **return**. Il ne met pas systématiquement fin à la méthode. Vous l'avez peut-être remarqué, mais *yield* signifie *céder*, donc à première vue on pourrait penser que **yield** met en pause l'exécution d'une méthode qui contient cette instruction, et c'est pas mal ce qui se passe. Prenons l'exemple suivant :

```
def rangeV2(debut, fin, pas):
    courant = debut
    while courant < fin:
        yield courant
        courant += pas

r = rangeV2(0, 10, 3)
print(r)
for i in r:
    print(i)
```

Ici, on se définie une méthode qui fait environ ce que **range**[8] fait : générer des nombres selon une valeur de début, de fin et un pas. Lorsqu'on exécute le code, la ligne `r = rangeV2(0, 10, 3)` n'exécute pas le code directement. Dans ce cas, `rangeV2` retourne ce qu'on appelle un `generator`. La compréhension de cet objet dépasse le but de ce guide, mais un lecteur curieux peut assez facilement trouver de l'information sur le web. Ce qu'il faut savoir, c'est que **yield** suspend l'exécution de `rangeV2` en gardant toutes ses variables internes en mémoire, retourne un objet `generator` et lorsqu'on itère sur ledit objet, c'est seulement là que le code est exécuté. Dans l'exemple précédent, la ligne **print(r)** n'afficher que la signature de l'objet `generator`, donc rien de bien utile. Lorsqu'on entre dans la boucle plus bas, on exécute une première fois le code et la console affiche 0, **yield** a cédé la variable `courant` qui valait 0. Puis, on retourne dans la boucle et la console affiche 3, **yield** à concédé 3. On retourne à la boucle, on voit que la console affiche 6, car c'était ce que valait `courant` lorsqu'on a atteint pour une troisième fois **yield**. Finalement, on retourne à la

8. Par contre, **range** ne donne pas d'objet `generator`. C'est un algorithme différent qui est utilisé.

boucle et cette fois-ci la console affiche 9, **yield** a donc donné la valeur de courant. Par la suite, le générateur est vide, car `courant` ne respecte plus la condition interne à `rangeV2`. On dit que le générateur est épuisé et on ne peut plus itérer dessus. Le processus d'exécution de `rangeV2` est terminé ! On peut conclure de cela qu'on ne peut itérer sur un générateur qu'une seule fois. Malgré cela, ils peuvent être très utiles lorsqu'on doit générer une énorme quantité de données. Elles sont générées une après l'autre et ne sont pas stockées ensemble, contrairement à une liste.

6.2 Récursion (à googler : récursivité)

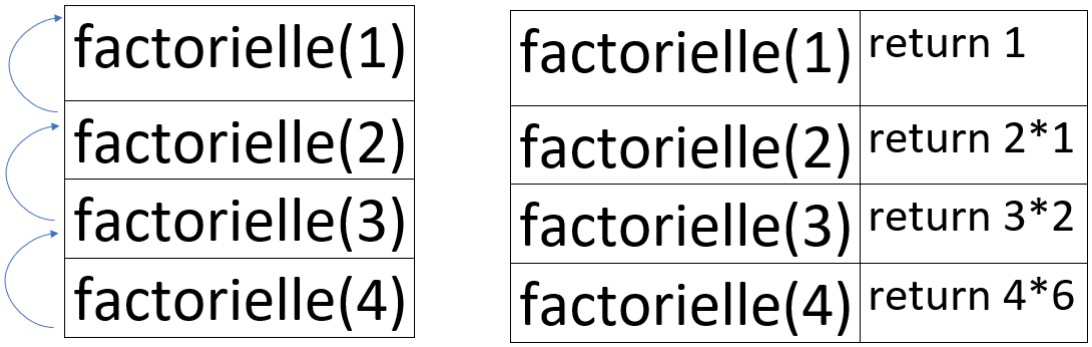
Concept naturel à plusieurs langages de programmation, notamment grâce à la manière dont les appels aux fonctions sont gérés, la récursion (ou récursivité, récurrence) est utile dans certains cas, soit pour enlever des boucles ou bien parce que le concept à modéliser est naturellement récursif, comme l'implémentation des graphes dans un programme informatique. Le concept sous-jacent à la récursion est relativement simple, quand on appelle une fonction, celle-ci est empilée sur la pile d'appels. Elle est seulement dépilée lorsqu'on sort de la fonction, soit par un **return** ou autre chose. Donc, si on rappelle cette fonction à partir d'elle-même, elle est rempilée par-dessus son appel précédent, et ainsi de suite. Cela nous permet parfois d'écrire un code plus compact. La récursion peut parfois causer certains problèmes, car elle peut ne jamais se terminer, comme une boucle. Il faut alors définir une condition d'arrêt qui met fin à la récursion et permet de dépiler les appels pour obtenir un résultat.

Prenons l'exemple de la factorielle :

```
def factorielle(n):
    if n % 1 != 0:
        return None # pas de factorielle de nombres non entiers
    if n < 0:
        return None # pas de factorielle negative
    if n <= 1:
        return 1 # 1! = 0! = 1
    return n * factorielle(n-1)
```

On remarque les conditions d'arrêts si n n'est pas entier, si n est négatif ou s'il est 0 ou 1. Sinon, on passe à la ligne du **return**. Il faut évaluer `n * factorielle(n-1)` avant de pouvoir retourner la valeur, donc on entre une nouvelle fois dans `factorielle`, mais avec $n-1$. Le cercle continu jusqu'à ce que $n-1=1$, ce qui permet de retourner 1, qui débloque le retour du niveau précédent, etc. Au final, on a $1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$.

On dit souvent qu'une image vaut 1000 mots, voici un schéma de ce qui se passe lorsqu'on veut calculer la factorielle de 4 :



(a) Empilement des factorielles

(b) Dépilement des factorielles lorsque les returns ont finalement retournés

FIGURE 6.1: Visualisation des factorielles

En revenant sur la figure 6.1(b), il faut comprendre que ce qui se passe est : `factorielle(1)` retourne 1, ce qui permet à `factorielle(2)` de retourner 2×retour de `factorielle(1)` (donc 2), ce qui permet à `factorielle(3)` de retourner 3×retour de `factorielle(2)` (donc 6), ce qui permet à `factorielle(4)` de retourner 4×retour de `factorielle(3)` (donc 24).En somme, on a que $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Le meilleur moyen pour bien comprendre la factorielle est de dessiner ce genre de chose. Ainsi, on visualise ce qui se passe et ça aide grandement à bien comprendre.

6.3 Plus à propos des arguments

Parler de *args et **kwarg

7 Importation et certains mots importants

7.1 Comment importe-t-on un ou plusieurs fichiers ?

En *Python*, comme dans tous les langages qui se respectent, les différents outils qui peuvent nous être utiles se retrouvent souvent dans d'autres fichiers ou dans des modules externes. Dans la plupart des projets, il est essentiel de faire appel à l'importation de fichiers. Pour se faire, *Python* utilise une nomenclature relativement simple, mais efficace. Il ne sera pas ici question d'installation de modules externes, mais bien d'importation dans un fichier *Python*. Pour y arriver, il y a quelques formulations importantes :

`import <nom de fichier>`	On importe le fichier
`import <nom de fichier> as <alias>`	On importe le fichier en lui donnant un alias
`from <nom de fichier> import <fonction, variable>`	On importe certains éléments du fichier
`from <nom de fichier> import *`	On importe tout ce qui peut être importé dans le fichier

7.2 Mots-clés importants

Il existe une certaine quantité de mots-clés en *Python*. Chaque mot-clé sert à une opération spécifique. Par exemple, **`import`** sert à l'importation. Cette sous-section sert à présenter quelques mots-clés qui ne seront pas présentés au fil de ce document, dans d'autres sous-sections plus spécifiques.

— `None` : représente une valeur nulle, ou tout simplement aucune valeur

— **`pass`** : à utiliser lorsqu'on ne met aucun code. Dans des langages où l'utilisation des accolades sert à délimiter les différents blocs de code, on peut simplement ne rien écrire. Par contre, en *Python*, il faut ajouter **`pass`**

— **`del`** : Peut-être utilisé de quelques manières différentes. Par exemple, il peut être utilisé pour supprimer l'élément d'une liste ou d'un dictionnaire. Il peut aussi être utilisé pour supprimer une variable entièrement, c'est-à-dire que si on refait appel à la variable, on va avoir une erreur, car la référence va être indéfinie. À faire attention !

— `with` : utilisé dans un *with statement*. Ce mot-clé est principalement utilisé lorsqu'on effectue une tâche et que, lorsque cette tâche est terminée, on doit faire une action spécifique et assez importante. Par exemple, lorsqu'on utilise les données d'un fichier, il faut premièrement lire le fichier, en extraire les informations et après fermer le fichier. Lorsque certaines méthodes sont implémentées, on peut utiliser le `with`. Par exemple, pour lire un ficher, on aurait :

```
with open("test.txt") as fichier:
    <code a executer>
```

et lorsqu'on quitte le *with statement*, l'action de fermer le fichier est effectuée.

8 Accès aux variables

Les modificateurs d'accès aux variables est un concept controversé au sein des développeurs *Python*. Certains vont dire que les utilisateurs et autres programmeurs sont assez matures pour ne pas jouer avec les variables internes, alors que d'autres, bien souvent venant de langages différents où on veut protéger à tout prix les variables internes, considèrent que privatiser les variables auxquelles il ne faut pas toucher est une pratique nécessaire. Peu importe dans quel camp, on se trouve, il existe des alternatives en *Python* pour ajouter un semblant de vie privée à nos variables, c'est _. On le connaît principalement pour son utilité en tant que délimiteur de mots dans le nom des variables ou des fonctions, mais lorsqu'il se trouve devant ou derrière un nom, il a un tout autre rôle. Nativement, il n'est pas possible de privatiser à 100% une variable ou une méthode, mais les développeurs de *Python* ont ajouté certaines possibilités. Si on met un seul souligné (*undescore*), cela signifie que l'accès à la variable est légèrement protégé. On peut y accéder, mais la plupart du temps c'est déconseillé, principalement parce que c'est une variable interne importante qui ne devrait pas être modifiée. Si on en met deux, cela signifie que la variable est privée, donc qu'on ne doit en aucun cas y accéder. Cela peut être une variable contenant un mot de passe, ou bien une variable interne qui ne doit pas être modifiée. Dans le cas du souligné simple, l'interpréteur *Python* ne se soucie pas vraiment de qui accède à la variable ou méthode, c'est plutôt un signe à l'utilisateur ou au développeur que le contenu de cette variable ne devrait pas être accédé ou modifié. Le souligné simple à un impact dans l'importation. Prenons le code suivant :

```
# fichier exemple.py
def ma_fonction():
    return 2

def _ma_fonction_2():
    return 2+2
```

```
# fichier utilisation_exemple.py
from exemple import * # on importe tout de exemple.py, sauf...
print(ma_fonction())
print(_ma_fonction_2())
```

La dernière ligne de l'exemple précédent va causer une exceptions, car l'importation avec $*$, qui importe tout le contenu d'un fichier, ne tient pas compte des fonctions commençant avec un ou deux soulignés. Par contre, si on procède avec :

```
# fichier utilisation_exemple.py
import exemple
print(exemple.ma_fonction())
print(exemple._ma_fonction_2())
```

On obtient la sortie :

```
2
4
```

Dans le cas du souligné double, l'interpréteur *Python* lance une exception lorsqu'on tente d'accéder à cette variable à partir d'une classe ou d'un fichier autre que celui dans lequel la variable est définie. Cela s'applique aussi aux méthodes. Il est à noter qu'on peut outrepasser cela, mais il est largement déconseillé de le faire. Si un développeur a décidé d'écrire `__ma_variable_privee`, c'est sûrement parce qu'il avait une bonne raison.

Vous avez peut-être remarqué que certaines fonctions *built-in* ont des doubles soulignés au début et à la fin, comme `__init__`, que vous avez probablement déjà rencontré (voir 11.2 pour plus d'information sur cette méthode). Cela signifie que ce sont des méthodes spéciales au langage *Python*. Il est déconseillé d'écrire un nom de variable ou de fonction ayant deux soulignés avant et après.

Finalement, que signifie un seul souligné, sans autres caractères avant ou après ? En *Python*, le nom `_` est souvent associé à une variable temporaire ou insignifiante. On peut l'utiliser dans une boucle simple :

```
for _ in range(1000):
    print("Toto")
```

Au final, ce sera à vous (ou votre superviseur, équipe de recherche, ...) de décider si vous incorporez les éléments de privatisation de variables.

9 Fonctions built-in

9.1 Les fonctions sum, min, max et abs

Certaines fonctions mathématiques de base sont déjà implémentées dans le langage python. En voici quelques-unes.

La somme.

En mathématique.

$$\sum_{0}^{9} i = 45$$

En python.

```
    array: list = [i for i in range(10)]
    array_sum = sum(array)
    print(array_sum)

out:
    45
```

Le minimum et le maximum.

```
    array_min = min(array)
    array_max = max(array)
    print(array_min)
    print(array_max)

out:
    0
    9
```

La valeur absolue.

```
    value = -7
    value_abs = abs(value)
    print(value_abs)

out:
    7
```

9.2 La fonction non définie lambda

La fonction non définie **lambda** est très utile en python pour sa simplicité et pour son efficacité. On sait qu'une fonction définie en python se fait de cette façon :

```
    def f(arg0, arg1, arg2, ...):
        return expression
```

Eh bien, la fonction **lambda** n'est pas bien plus compliquée. Elle prend n'importe quel nombre d'arguments et retourne une expression simple. Sa syntaxe est alors,

```
    lambda arguments: expression
```

Voici un exemple.

```
    f = lambda x: 2*x + 1
    print(f(2))
out:
    5
```

Un exemple avec plusieurs arguments.

```
    g = lambda x, y, z: x + y + z
    print(g(1, 2, 3))
out:
    6
```

Un exemple avec un objet en argument.

```
    h = lambda iterable: sum(iterable)
    print(h([i for i in range(10)]))
out:
    45
```

Maintenant, regardons une autre utilité de la fonction lambda, les arguments non définis. Prenons par exemple une certaine fonction Y.

```
    def Y(m: float, b: float):
        return lambda x: m*x + b
```

Et maintenant, nous voulons créer une sous-fonction ayant les paramètres $m = 5$ et $b = 1$.

```
    y = Y(5, 1)
    print(type(y))
out:
    <class 'function'>
```

Nous avons qu' y est une fonction lambda avec les paramètres m et b passés à la fonction Y. Alors,

```
    print(y(2))
out:
    11
```

9.3 Autres fonctions

Beaucoup d'autres fonctions très pertinentes sont disponibles en python. Vous pourrez en voir une liste plus détaillée sur ce site :

https://www.w3schools.com/python/python_ref_functions.asp

10 Exercices (récapitulatifs jusqu'à maintenant)

Les exercices suivants peuvent assez fortement ressembler à des exercices d'examens des cours d'introduction à la programmation. Les solutions (détaillées et expliquées pour certains exercices) se trouvent à la fin.

10.1 Calculer un polynôme

Un polynôme d'ordre n est défini comme suit :

$$p(x) = a_0 + a_1 x^1 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i$$

On vous demande d'écrire une fonction qui prend deux paramètres : une liste de coefficients a_i et une valeur de x. Vous pouvez prendre la signature `polynome(coefficients, x)`. Si on vous donne `coefficients = [0, 1, 2, 3]` et `x = 3`, vous devriez retourner $0 \cdot 3^0 + 1 \cdot 3^1 + 2 \cdot 3^2 + 3 \cdot 3^3$.

10.2 Conversion d'unités

On vous demande de créer trois fonctions distinctes permettant de convertir des pouces en cm, des degrés Celsius en degrés Fahrenheit et des terrains de football (américain) / année (non bissextile) en km / heure. Voici quelques infos pertinentes :

— 1 po = 2.54 cm
— $x^\circ F = \frac{5(x-32)}{9}^\circ C$
— Longueur d'un terrain de football = 91.44 m
— Une année (non bissextile) = 365 jours, 1 jour = 24 heures

10.3 Entropie en théorie de l'information

Dans le fabuleux domaine de la théorie de l'information, on peut définir l'entropie [9] (aussi appelée entropie de Shannon, similaire à l'entropie de Gibbs en mécanique statistique) :

$$H = -\sum_{i=0}^{n} p_i \ln p_i$$

où p_i représente une probabilité, par exemple la probabilité que la valeur x_i apparaisse dans un ensemble X de valeurs quelconque. Cette probabilité s'écrit comme :

$$p_i = \sum_{j=0}^{n} \frac{\delta_{x_i, x_j}}{|X|}$$

9. Dans ce contexte-ci, l'entropie est en quelque sorte une mesure de prédictibilité, car plus la valeur est élevée, moins on serait en mesure de prédire la valeur de l'élément, alors que si l'entropie est faible, il est plus facile de prédire. Dans le cas d'une entropie nulle, l'ensemble est composé d'un seul élément, ou bien tous les éléments ont la même valeur.

où δ_{x_i,x_j} est le delta de Kronecker et $|X|$ est la cardinalité de l'ensemble (i.e. le nombre d'éléments).

On vous demande d'écrire une fonction qui prend en argument un objet itérable (liste ou tuple, tous en 1D) et qui retourne la valeur d'entropie associée à cet itérable (indice : ces objets itérables possèdent une méthode qui peut vous être très utile pour compter le nombre d'occurrences d'une valeur).

10.4 Entropie dans l'ensemble microcanonique de la mécanique statistique

Vous verrez plus tard, dans le cours de *Physique statistique* ce qu'est l'ensemble microcanonique, mais pour l'instant, il est seulement important de savoir qu'il s'agit d'un système où les états possibles sont équiprobables, donc qu'ils ont la même probabilité. Par exemple, dans un système avec 3 états possibles, chaque état a une probabilité de 1/3.

Sachant cela, on vous demande d'écrire une fonction génératrice qui prend en argument une liste de valeurs correspondant au nombre d'états accessibles pour un système (une valeur comme le 3 de l'exemple un peu plus haut) et qui retourne (`yield`) l'entropie associée. L'entropie dans l'ensemble microcanonique est donnée par :

$$S_i = k_B \ln \Omega_i$$

où $k_B = 1.380649 \times 10^{-23}\ J/K$ est la constante de Boltzmann et Ω_i est le nombre d'états accessibles, soit la i^e valeur de l'ensemble passé en argument. Voici un exemple d'exécution : si on vous donne la liste `[2]`, cela veut dire que le système ne peut être que dans deux états distincts. Dans ce cas, l'entropie est $k_B \ln 2$, donc la fonction devrait retourner $9.56992629 \times 10^{-24}$. Si on vous donne `[1,2]`, vous devriez retourner 0 et $k_B \ln 2$ (en valeur numérique bien sûr !)

10.5 Permutation de lettres dans une chaîne de caractères

On vous demande d'écrire une fonction qui prend en argument une chaîne de caractères et qui retourne le nombre de permutations de lettres que l'on peut faire avec cette chaîne. Une permutation est tout simplement le fait d'échanger deux lettres dans une chaîne de caractères. De manière mathématique, on peut écrire :

$$P = \frac{n!}{r_1! r_2! \cdots r_k!}$$

où n est le nombre de lettres dans la chaîne et k représente le nombre de caractères distincts dans la chaîne. Par exemple, avec la chaîne `"elle"`, il y a $6 = \frac{4!}{2!2!}$ permutations possibles. La fonction `factorial` de `math` pourrait vous être utile.

10.6 Renverser l'ordre des lettres dans une chaîne de caractères

On vous demande d'écrire une fonction qui prend en argument une chaîne de caractères et qui renverse l'ordre des caractères et qui retourne la nouvelle chaîne. Par exemple, si on vous donne `"Python"`, vous devriez retourner `"nohtyP"`. (Indice : utiliser des slices peut donner le bon résultat en une seule ligne. Sinon, il se peut que vous ayez à utiliser `join` de la classe **`str`**)

10.7 Nombre d'or : fraction continue vs racine continue

Le nombre d'or est une proportion qui, entre autres, est le seul rapport a/b entre deux longueurs a et b respectant :

$$\frac{a+b}{a} = \frac{a}{b}$$

où a est la plus grande longueur des deux. De plus, c'est l'unique solution de l'équation :

$$x^2 - x - 1 = 0$$

Ce nombre est défini de manière exacte par :

$$\phi = \frac{1+\sqrt{5}}{2}$$

On vous demande d'écrire deux méthodes distinctes approximant le nombre d'or. Elles prendront un seul paramètre, soit le nombre d'itérations pour l'approximation. D'une part, vous devrez utiliser l'approximation de la fraction continue :

$$\phi = \frac{1}{1+\frac{1}{1+\frac{1}{1+\ldots}}}$$

D'autre part, vous devrez utiliser l'approximation de la racine continue :

$$\phi = \sqrt{1+\sqrt{1+\sqrt{1+\ldots}}}$$

Veuillez noter que l'utilisation de la récursion s'applique très bien à ces deux problèmes. De plus, les résultats peuvent être variés, donc si vous comparez les réponses avec d'autres solutions, il est normal que les solutions diffèrent.

11 La programmation orientée objet

11.1 Utilité et avantage

La programmation orientée objet (POO ou OOP pour les intimes) est un paradigme développé durant les années 60 et 70. Ce paradigme consiste principalement en la définition de blocs de code logiquement liés entre eux qu'on appelle objets. La programmation orientée objet est possible dans les plus grands langages de programmation, à l'exception du langage *C*. En effet, on peut en faire en *C++*, *Java*, *Python*, *C#* pour ne nommer qu'eux. Un grand avantage de ce paradigme est probablement le fait qu'on regroupe, dans ce qu'on appelle une classe, toutes les variables et fonctions qui ont un lien logique envers une entité quelconque qu'on désire modéliser grâce à la programmation. Considérez l'exemple suivant : je désire modéliser un chat dans un programme donnant à l'utilisateur le contrôle du félin. Sachant qu'un chat a quatre pattes (normalement) et qu'il ronronne, ce serait logique de se créer un genre de structure de données permettant d'avoir une variable/constante contenant le nombre de pattes et une fonction qui permet au chat de ronronner. C'est dans ce genre de cas que la POO est très utile et efficace. On aurait donc à écrire :

```
class Chat:
    def __init__(self):
        self.nombre_pattes = 4

    def ronronner(self):
        print("RRRrrrRRRrrr")
```

Ensuite on peut facilement utiliser ce qu'on vient d'écrire en se créant une variable qui sera du type `Chat` :

```
if __name__ == '__main__':
    chat = Chat()
    print(chat.nombre_pattes)
    chat.ronronner()
```

La sortie donne :

```
4
RRRrrrRRRrrr
```

Grâce à cet exemple simpliste, on voit que l'orienté objet permet essentiellement de regrouper des variables et des fonctions dans une structure logique pouvant modéliser toute sorte de concept physique, autant concret, comme un chat, qu'abstrait. Un autre avantage de la POO, c'est sa facilité d'entretien. Quand on a une classe et qu'on veut ajouter une méthode, et bien on l'ajoute. Dans certains cas qui ne font pas appel à la POO, il pourrait être beaucoup plus laborieux d'ajouter des méthodes, voire des attributs. On peut aussi dire que la POO aide à diminuer la redondance. Si je veux créer 15 chats ayant, par exemple, tous une couleur différente, je peux ajouter à ma classe un constructeur qui prend certains paramètres, dont une couleur de poils, qui peuvent rendre les objets différents entres eux, alors que si je ne fais pas de classe, je dois écrire 15 bouts de code différents.

Une classe peut être très complexe, ayant plusieurs attributs et méthodes, mais dans certains cas il est encore possible de factoriser en plus petites entités. Pour cela, on peut bien sûr créer d'autres classes indépendantes, mais on peut aussi utiliser l'héritage, mais avant de parler de ce concept, il est très important de parler du constructeur, mais avant...

11.1.1 Self, ou comment faire référence à l'objet courant

En POO, on dit d'un objet qu'il est une instance de classe, donc qu'un objet est le résultat spécifique de l'appel à une classe. Lorsqu'on écrit

```
chat = Chat()
ma_chaine = "hello world"
mon_complex = 4j + 3
```

on ne fait que créer des cas spécifiques de classes. En effet, la variable (ou objet) `chat` est un cas spécifique de la classe `Chat`[10], la chaîne "hello world" est un cas spécial de l'objet **`str`** et $4j+3$ est un instance de la classe **`complex`**. Comme une classe se doit d'être relativement générale, il ne faut pas que chaque attribut soit statique[11]. Pour se faire, on utilise un mot réservé par le langage : **`self`**. Ce mot est utilisé lorsqu'on a besoin d'une référence envers l'objet courant. Expliquons le principe d'objet courant à l'aide de la classe `Chat`, dont nous allons complexifier un peu :

```
class Chat:

    def __init__(self):
        self.nombre_pattes = 4
        self.longueur_poils = "long"

    def raser(self):
        self.longueur_poils = "court"
```

On peut remarquer que le paramètre **`self`** de la méthode `raser` est utilisé à même la méthode. De cette manière, seul l'objet courant subira la modification. Ainsi, si on utilise le code de la manière suivante :

```
grosminet = Chat()
jerry = Chat()
print(grominet.longueur_poils)
print(jerry.longueur_poils)
jerry.raser()
print(grominet.longueur_poils)
print(jerry.longueur_poils)
```

10. Dans le cas présent, même si chaque objet créé par la classe est le même dans le sens où ils ont les mêmes attributs, ils sont différents *à l'interne* de l'ordinateur, notamment par l'entremise qu'un adressage mémoire différent.

11. En programmation, on parle d'attribut ou de méthode statique lorsque : l'attribut est le même pour chaque objet crée ou lorsque la méthode est indépendante des objets créés.

Et le output est :

```
longs
longs
courts
longs
```

Donc, car la méthode *connaît* l'objet courant (dans ce cas, `jerry`), seul l'attribut `poil` de l'objet concerné est modifié. Les attributs de `grominet` ne sont pas touchés. Par contre, considérons l'exemple suivant :

```
class Chat:

    statique_compter_nb_chats = 0

    def __init__(self):
        self.nombre_pattes = 4
        Chat.statique_compter_nb_chats += 1

    def __del__(self):
        Chat.statique_compter_nb_chats -= 1
```

L'attribut `statique_compter_nb_chats` est ce qu'on appelle un attribut statique, donc qu'il a la même valeur pour toutes les instances qui seront créées jusqu'à ce que le programme s'arrête. Voici un exemple :

```
jerry = Chat()
print(jerry.statique_compter_nb_chats)

grominet = Chat()
print(grominet.statique_compter_nb_chats)

del grominet
print(jerry.statique_compter_nb_chats)
```

On obtient :

```
1
2
2
1
```

On remarque donc que la variable augmente lorsqu'on crée une nouvelle instance de l'objet, alors qu'elle diminue lorsque supprime une instance de l'objet. Ainsi, la valeur de la variable est la même pour toutes les instances de l'objet. On peut aussi utiliser le même concept et l'appliquer aux méthodes des classes. En ayant des méthodes statiques, elle est la même pour tout objet créé par la classe. On peut y arriver avec un décorateur, `@staticmethod`, et une telle méthode n'a pas besoin du paramètre `self`, mais on peut tout de même lui en passer. Prenons l'exemple suivant :

```
class Chat:
    def __init__(self):
        self.nombre_pattes = 4

    @staticmethod
    def cri(aFaim):
        return "MEOW" if aFaim else "meow"
```

Si on utilise le code de la manière suivante :

```
jerry = Chat()
grominet = Chat()

print(jerry.cri())
print(grominet.cri(True))
```

```
meow
MEOW
```

L'avantage d'avoir une méthode statique peut être qu'on n'a pas besoin d'un objet spécifique (comme faire une méthode utilitaire qui ne touche pas à la mécanique interne de l'objet), ou bien c'est un comportement commun à toutes les instances (comme dans l'exemple ci-dessus).

11.2 Le constructeur

Le constructeur est un principe fondamental en POO. L'idée derrière la POO est de créer des objets qui peuvent être différents, et ce sans avoir à réécrire le code à chaque fois. Le constructeur permet de faire cela. C'est une fonction spéciale[12] qui est utilisée une seule fois dans la vie des objets, soit à la création de celui-ci. Il n'est pas nécessaire de le définir, de l'écrire, car il en existe un par défaut, mais il est pas mal moins intéressant puisqu'il ne fait rien de spécial, il ne crée qu'un objet par défaut, sans paramètres. Les langages permettant l'orienté objet permettent tous l'implémentation d'un constructeur, mais certains langage requiert une attention particulière à certaines variantes du constructeur. En *Python*, la méthode `__init__`, qui fait office de constructeur, reste simple à utiliser. Elle prend au moins un paramètre, **self**, qui est une référence de l'objet courant utilisé ou qu'on est en train de créer dans le cas du constructeur. Ensuite, on peut ajouter autant de paramètres que l'on veut.

Considérez l'exemple suivant : on désire écrire un programme pour une grande chaîne de vêtements. La chaîne nous demande d'inclure des manteaux dans le logiciel. Pour ce faire, nous devons créer une classe, la classe `Manteau`.

```
class Manteau:
    def __init__(self, couleur, taille, en_fourrure=False):
        self.couleur = couleur
        self.taille = taille
        self.en_fourrure = en_fourrure
```

12. Ou *spétiale* selon Pr. Sheng

```
...
```

Vous remarquerez que le dernier paramètre est différent des autres. En fait, lorsqu'on écrit `param=<valeur>`, on assigne à ce paramètre une valeur par défaut. Donc, au moment où l'on fait appel au constructeur, on peut omettre de donner une valeur à ce paramètre si celui par défaut fait l'affaire. De plus, il est important de savoir que lorsqu'on utilise les paramètres avec valeur par défaut, il faut qu'ils soient à la fin de la liste de paramètres.

11.3 Héritage

L'héritage est un aspect très important et hautement intéressant de la programmation orientée objet. Ce concept est réellement lié au concept d'héritage comme on le connaît chez les humains. En effet, on a une classe mère qui contient une certaine quantité d'information, puis des classes dérivées de cette classe mère. Ces classes dérivées reprennent l'information de la classe parente, la modifient ou en ajoutent de la nouvelle. Dans le cas de *Python* et de certains autres langages, on peut parler d'héritage multiple, donc on a plusieurs classes mères. Dans ce cas, il faut faire attention à certains conflits. La plupart du temps, l'héritage simple est suffisant.

Reprenons l'exemple de la classe `Chat`, mais en changeant la structure en ajoutant une classe mère :

```
class Animal:
    def __init__(self, couleur, nombre_pattes):
        self.couleur = couleur
        self.nombre_pattes = nombre_pattes


class Chat(Animal):
    def __init__(self, couleur):
        Animal.__init__(self, couleur, 4)

    def ronronner(self):
        print("RRRrrrRRRrrr")
```

Grâce à l'héritage, on peut factoriser les problèmes en plein de sous-classes. Dans l'exemple précédent, on crée une classe mère qui contient des informations, puis le classe `Chat`, qui hérite de `Animal`, on utilise le constructeur de la classe parente en spécifiant un nombre de pattes. On ajoute aussi la fonction `ronronner`. Les possibilités sont immenses.

11.3.1 La redéfinition

La redéfinition de méthode est un concept qui peut être très important dans l'héritage. Si une classe parente définie une méthode d'une certaine manière, mais qu'une classe enfant à besoin de la modifier, c'est faisable. Considérez l'exemple suivant, on reprend la classe `Chat` :

```
class Animal:
    def __init__(self, couleur, nombre_pattes)
```

```
        self.couleur = couleur
        self.nombre_pattes = nombre_pattes

    def cri(self):
        pass


class Chat(Animal):
    def __init__(self, couleur):
        Animale.__init__(self, couleur, 4)

    def cri(self):
        print("meow")
```

On peut remarquer que la classe `Chat` redéfini la méthode `cri` et permet donc à chaque objet chat de miauler lorsqu'elle est appelée.

12 Les décorateurs

12.1 Qu'est-ce que c'est ?

Les décorateurs peuvent, à première vue, sembler sortir de nulle part. Il s'agit plutôt d'un moyen rapide et élégant d'effectuer des manipulations sur une fonction. Un aspect important de *Python* est que ses fonctions sont des objets au même sens qu'un `float` ou qu'un objet d'une classe quelconque. Ainsi, il est possible d'utiliser une fonction comme argument d'une autre fonction. Ainsi, il est possible de faire :

```
def fonctionInterne():
    print("Fonction interne")

def fonctionExterne(fonction):
    fonction() # On fait appel a la fonction avec les parentheses
```

Puis, plus loin :

```
fonctionExterne(fonctionInterne)
```

On aurait comme sortie en console :

```
Fonction interne
```

Il faut faire attention à ceci : l'ajout ou non de parenthèses lorsqu'on donne la fonction comme argument. La différence peut paraître subtile à première vue, mais elle est très important. Dans le cas où l'on ne met pas de parenthèse, l'argument se trouve à être l'endroit où se trouve la fonction. Ainsi, c'est une référence envers la fonction. Par contre, lorsqu'on ajoute les parenthèses, on appelle la fonction, donc elle exécute son code interne. Dans l'exemple précédent, si on affichait ce qu'est l'argument `fonction` dans le corps de la fonction, on verrait quelque chose qui ressemble à :

```
<function fonctionInterne at 0x000002139A0062F0>
```

Puis, en ayant cette référence, il est possible d'appeler la fonction à l'aide d'ajout des parenthèses à la référence, comme ce qui est fait à l'intérieur de `fonctionExterne`. Si, par mégarde, on ajoutait les parenthèse lors de l'appel à `fonctionExterne` (i.e. on fait `fonctionExterne(fonctionInterne())`), on ne donnerait pas la référence en argument, mais plutôt le résultat de ce que `fonctionInterne` fait (ici, ce serait `None` donc il y aurait une erreur lors de l'exécution, car un objet `None` ne peut être appelé par les parenthèses). Un autre point important est que comme une fonction peut être vue comme une variable (de part sa référence), on peut donc retourner une fonction ! En effet, cela est possible. Essayez par vous-mêmes et vous verrez que les fonctions sont très malléables.

12.2 Faire ses propres décorateurs

Maintenant qu'on connaît l'appel aux fonctions à l'aide de leur référence, on peut mieux comprendre la beauté et l'utilité des décorateurs. Vous vous demandez peut-être qu'est-ce qu'un décorateur ? La réponse simple est : c'est une fonction qui prend une fonction et effectue quelque chose avec. Vous avez sûrement déjà vu un décorateur. Les plus courants sont sûrement `@staticmethod` et `@classmethod` lorsqu'on travaille avec la programmation orientée objet. Ici, le symbole `@` signifie que ce qui vient est le nom d'un décorateur (le nom d'une fonction). Puis, il est implicite que ce qui suit l'appel au décorateur est une déclaration de fonction qui sera donnée en argument au décorateur. Dans sa forme la plus simple, on procède comme suit :

1. On crée une fonction (fonction décoratrice) qui prend un seul argument : la fonction qu'on veut décorer (fonction décorée)
2. On crée une fonction interne (fonction interne) à la fonction décoratrice qui s'occupe des arguments de la fonction décorée.
3. La fonction interne s'occupe d'au moins appeler la fonction décorée avec ses arguments, puis elle peut faire autre chose en plus (afficher une message, compter, etc.)
4. La fonction interne retourne (au moins !) le retour de la fonction décorée
5. La fonction décoratrice retourne la fonction interne (sa référence)

En pseudo-code, ça ressemble à :

```
def decorateur(fonction):
    <faire quelque chose si voulu>
    def wrapper(*args, **kwargs):
        <faire quelque chose si voulu>
        f = fonction(*args, **kwargs)
        <faire autre chose>
        return f
    <faire quelque chose avant le retour>
    return wrapper
```

Si vous avez besoin de vous souvenir de ce que `*args` et `**kwargs` font, allez voir la section 6.3. Si vous ne voulez qu'un simple rafraîchissement de mémoire, `*args` est simplement un raccourci si on ne connaît pas totalement les arguments voulus, alors que `**kwargs` permet aussi ce genre de solution, mais en utilisant des mots clés (comme dans `fonction(a=2)`, où on assigne `a` directement par son mot clé). Bref, cela nous assure d'avoir tous les arguments possibles, sans en manquer un. En se référant à l'exemple de pseudo-code, on remarque qu'un décorateur est en fait des fonctions imbriquées qui utilisent une autre fonction (et possiblement d'autres arguments). La compréhension de la logique derrière (comment `@staticmethod` fait en sorte de prendre la référence à la méthode définie juste après, pourquoi on doit avoir une fonction imbriqué, pourquoi on retourne la référence de la fonction interne) n'est pas très importante : des gens l'ont fait comme ça, ainsi soit-il. Vous pouvez tout de même trouver des ressources plus techniques sur le web, mais nous nous concentrerons sur des implémentations intéressantes et permettant plus de diversité au lieu de s'attarder sur des concepts qui risquent de perdre votre intérêt. Passons à un exemple concret qui pourrait vous être utile. On dit souvent qu'on est jamais mieux servi que par sois-même, alors au lieu d'utiliser une chronomètre pour compter combien de temps prend l'exécution d'une fonction, on va s'en faire un ! Premièrement, on doit penser à la structure qu'on veut utiliser. En *Python*, on

peut simplement se faire un fichier et y mettre nos fonctions dedans, sans se soucier de faire une classe. Voici comment on pourrait se faire un chronomètre :

```
import time # On veut acceder au temps!

def timerBase(fonction, *fargs, **fkwargs):
    debut = time.time() # retourne le temps a cet instant
    f = fonction(*fargs, **fkwargs)
    fin = time.time()
    return f, fin - debut # On retourne l'exec de la fonction + temps pris

def timer(fonction):
    def wrapper(*args, **kwargs):
        f, temps = timerBase(fonction, *args, **kwargs)
        return f, temps
    return wrapper
```

Puis, essayons le code :

```
<... dans le meme fichier, plus bas...>

@timer # On fait appel a notre decorateur
def test():
    time.sleep(3) # Fait "dormir" le processus pendant 3 secondes

if __name__ == "__main__":
    print(test())
```

On aurait comme sortie :

```
(None, 3.007863998413086)
```

Attention, bien qu'on fasse dormir pendant 3 secondes, il se peut (comme ici) que le chronomètre n'affiche pas 3 parfaitement, car il se peut que d'autres processus soit survenus, ou bien qu'il y ait de la latence provenant du code interne à `time.sleep`. Retournons maintenant décortiquer le code défini dans l'exemple du chronomètre. Premièrement, on importe le module `time`. Ce module est utilisé lorsqu'on veut accéder à l'horloge interne de l'ordinateur. Il est assez basique, contrairement à ce que `datetime` peut offrir, mais ses méthodes sont amplement suffisantes pour ce qui nous intéresse, de plus sa simplicité peut être avantageuse, car plus le module est simples, plus il est *rapide* dans ce qu'il a à faire. Dans notre cas, ce n'est pas super critique d'avoir un peu de latence, car il s'agit d'un exemple simple, mais dans certains cas où il faut avoir une grande précision, chaque milliseconde (voire micro ou nano !) est importante ! Après avoir importé le nécessaire, on se défini une fonction qui prend (au minimum) un argument : la fonction qu'on veut chronométrer. On se souvient que `*fargs` et `**fkwargs` sont simplement là pour permettre l'accès à tous les paramètres possibles de la fonction passée en argument. Cette fonction n'est pas nécessaire, car on peut prendre son code et le mettre directement dans le décorateur, mais on a procédé ainsi pour factoriser le code et le rendre plus facile à lire. La première ligne du corps de la fonction est simplement un appel au *temps* courant en secondes. En effet, `time.time()` retourne le temps en secondes écoulé depuis une

certaine date. La plupart du temps, cette date est le 1er Janvier 1970 à 00 : 00 : 00 [13], mais il peut varier selon la machine. Par contre, que cette date soit le 1er Janvier 1970 ou le 23 juin 1987, cela n'a pas d'importance dans notre cas, car on ne s'intéresse qu'à la différence entre deux appels [14]. Maintenant qu'on a une référence sur le temps de départ, on peut faire appel à la fonction en lui fournissant les arguments nécessaires. On garde en mémoire le retour de la fonction, car il peut être important de savoir ce qui est retourné. Avant de quitter la fonction de chronomètre, on accède au temps après l'exécution de la fonction. Finalement, on retourne le retour de l'exécution ainsi que le temps pris. L'avantage d'avoir procéder de cette manière est non seulement pour la lecture du code comme mentionné plus haut, mais aussi pour la possibilité d'utiliser le chronomètre sans décorateur. En effet, il est possible d'utiliser cette fonction sans passer par la décoration, car on peut directement passer par cette fonction. En reprenant l'exemple précédant, on peut simplement faire :

```
def test_sans_decorateur():
    time.sleep(3)

if __name__ == "__main__":
    print(timerBase(test_sans_decorateur))
```

Et on a comme sortie

```
(None, 3.000436544418335)
```

Le temps peut être plus court pour plusieurs raisons, notamment par le fait qu'on utilise moins de code que lorsqu'on passe par le décorateur, mais en faisant cela on perd aussi son avantage qu'est la simplicité. Il est plus beau et clair de passer par un décorateur.

En se penchant maintenant sur le code permettant d'utiliser le chronomètre comme décorateur, on remarque la structure imbriquée présentée en début de section : on a une fonction *globale* qui prend la référence vers la fonction à chronométrer, puis on a une fonction *locale* qui s'occupe des arguments et qui s'occupe de calculer le temps ainsi que d'exécuter la fonction. Cette fonction imbriquée retourne ensuite l'exécution de la fonction ainsi que le temps pris. Finalement, la fonction *globale* retourne la référence vers celle *locale*. Il n'est pas nécessaire de comprendre la nature même de l'algorithme, mais comprendre comment faire un décorateur et connaître la structure à utiliser est ce que nous voulions montrer.

13. Cela peut être différent selon le système d'exploitation. Si le lecteur veut vérifier quelle est sa date de référence, utiliser `time.gmtime(0)` donne toute l'information relative à l'heure zéro de l'ordinateur.

14. Dans ce cas, utiliser `time.time()` est purement arbitraire, car il serait aussi possible d'utiliser `time.monotonic()` ou `time.perf_counter()`. Ces deux méthodes ont une référence indéfinie, alors seulement une différence entre les résultats est pertinente. Il existe aussi les variantes retournant un temps en nanosecondes.

12.3 Décorateurs plus généraux et polyvalents

Il est peut-être important dans certains cas d'être capable de donner des arguments aux décorateurs, sans nécessairement passer par la fonction décorée. Par exemple, on pourrait afficher un message à l'écran, enregistrer dans un fichier si on donne un nom, etc. Dans notre cas, on va afficher une message en console. Pour obtenir un décorateur prenant un (voire plus) argument, on doit ajouter une épaisseur de fonction. On se retrouve alors avec la structure suivante :

```
def decorateur(argument1, argument2):
    <faire quelque chose si voulu>
    def fonction_interne(fonction):
        <faire quelque chose si voulu>
        def fonction_interne_interne(*args, **kwargs):
            f = fonction(*args, **kwargs)
            <faire autre chose>
            return f
        return fonction_interne_interne
    return fonction_interne
```

Maintenant, faisons un chronomètre prenant un message en paramètre :

```
import time

def timerBase(fonction, *fargs, **fkwargs):
    debut = time.time()
    f = fonction(*fargs, **fkwargs)
    fin = time.time()
    return f, fin - debut

def timerWithMessage(message):
    def timer(fonction):
        def wrapper(*args, **kwargs):
            f, temps = timerBase(fonction, *args, **kwargs)
            print(f"{message}")
            return f, temps
        return wrapper
    return timer
```

Puis, on peut tester le décorateur :

```
<... dans le meme fichier, plus bas...>

@timerWithMessage("Dort environ 3 secondes")
def test():
    time.sleep(3)

if __name__ == "__main__":
    test()
```

Et on a la sortie en console :

```
Dort environ 3 secondes
```

Voici ce qui fait pas mal le tour d'une introduction pratique aux décorateurs *Python*. Nous espérons que vous voyez leur utilité de part leur relative simplicité autant algorithmique que pratique (on les appelle facilement avec `@`!). On peut les utiliser pour surveiller du code comme ce qui a été fait dans les exemples du chronomètre. On pourrait aussi suivre la gestion de la mémoire en mesurant le nombre d'octets utilisés ou bien on pourrait aussi les utiliser pour enregistrer des données dans un fichier externe. Comme on dit, *sky is the limit* ou plutôt en programmation *RAM is the limit*[15].

Les décorateurs ne sont pas essentiels à bien programmer, autant d'un point de vue scientifique que non. En effet, il est plus important pour un scientifique d'apprendre à bien vectoriser son code ou à bien utiliser les divers outils scientifiques que de savoir comment faire un décorateur ! Par contre, faire son propre décorateur peut être très satisfaisant et élégant. Mettre des petits `@timer` ou `@memoryMonitoring` est très satisfaisant et esthétiquement plaisant, surtout quand nous en sommes l'auteur !

15. Les auteurs ne savent pas si cette expression existe vraiment. Si oui, tous droits réservés à son auteur.

13 Modules importants

13.1 NumPy

Le module *NumPy* est probablement le plus important lorsqu'on sort du cadre d'un cours d'introduction à la programmation. Il ajoute son lot d'avantages. Il ajoute de nouveaux objets extrêmement utiles, notamment les objets `ndarray` qui sont des tableaux statiques [16] pouvant contenir n dimensions (d'où le *nd*). Les *arrays* sont des objets très polyvalents pouvant contenir toute sorte d'objets, que ce soit des `str`, des `float` ou bien des objets créés par l'utilisateur. Il faut par contre que tout le contenu du tableau soit de la même sorte.

13.1.1 Installation

L'installation de *NumPy* se fait souvent lorsqu'on installe un interpréteur *Python*. Par exemple, si on télécharge *Anaconda*, le module devrait venir directement avec. Par contre, si il n'est pas présent, il est assez facile de l'installer, notamment grâce aux outils comme `pip` ou encore l'outil *Anaconda Navigator*. Voici un lien pour l'installer (de même que d'autres packages très pertinents dans le monde scientifique) :
`https://www.scipy.org/install.html`

13.1.2 Les premiers pas

Pour utiliser *NumPy* dans son code, c'est très simple. Il suffit de l'importer (voir section 7). Habituellement, on fait `import numpy as np`, car lui donner un alias permet d'avoir à écrire moins de choses. À la racine du module se trouve une panoplie de fonction statique qu'on peut utiliser, pour la majorité, avec plein de types. On peut bien sûr les utiliser avec des *arrays*, mais *NumPy* apporte aussi la notion de *array-like*, ce qui est assez large. Il n'y a pas vraiment de définition formelle de ce que c'est, par contre presque tout peut être traité comme un *array-like*.

Considérons l'exemple suivant :

```
import numpy as np

angle = np.arcsin(3 ** 0.5 / 2) # = pi / 3
```

On donne à la fonction un réel, ce que la fonction considère comme un *array* de dimension 0. Ainsi, le module n'apporte pas que de nouvelles fonctions ne fonctionnant qu'avec des *arrays*, mais bien des fonctions très générales qui peuvent être utilisées dans tous les types de projets, même ceux qui ne demandent pas l'utilisation de tableaux ! Il n'est pas rare de voir le module `math` remplacé par *NumPy*. Maintenant, comment créer un fameux `ndarray`. Il existe plusieurs manières d'instancier un *array*, en voici quelques-unes. [17] :

16. On les dit statiques, car contrairement à l'objet `list`, on ne peut changer sa taille sans avoir à définir un nouvel objet.

17. À noter qu'il existe plus d'arguments pouvant être entrés en paramètres. Ceux présentés sont considérés comme les plus pertinents pour l'instant.

— Passer par `np.zeros(shape, dtype)` : on crée un tableau de forme `shape` (un entier ou un tuple d'entiers) dont les objets seront du type `dtype`. Le tableau sera uniquement composé de 0 (ou `''` si c'est un tableau de chaînes de caractères). Exemple :

```
import numpy as np

zeros = np.zeros((2,2), float) # On cree un tableau 2x2 de floats
```

— Passer par `np.ones(shape, dtype)` : on crée un tableau de forme `shape` (un entier ou un tuple d'entiers) dont les objets seront du type `dtype`. Le tableau sera uniquement composé de 1 (ou `'1'` si c'est un tableau de chaînes de caractères). Exemple :

```
import numpy as np

ones = np.ones((2,2), float) # On cree un tableau 2x2 de floats
```

— Passer par `np.empty(shape, dtype)` : on crée un tableau de forme `shape` (un entier ou un tuple d'entiers) dont les objets seront du type `dtype`. Les entrées du tableau ne seront pas initialisées, ce qui peut permettre de sauver un peu de temps de calcul. Exemple :

```
import numpy as np

empty = np.empty((2,2), float) # On cree un tableau 2x2 de floats
```

— Passer par `np.full(shape, remplissage, dtype)` : on crée un tableau de forme `shape` (un entier ou un tuple d'entiers) dont les objets seront du type `dtype`. Les entrées du tableau seront initialisées à la valeur de `remplissage`. Exemple :

```
import numpy as np

full = np.full((2,2), np.pi, float) # On cree un tableau 2x2 de floats
# Chaque entree contient la valeur de pi
```

Si on dispose initialement d'un *array* ou d'un autre objet itérable (étant compris dans l'ensemble *array-like*), on peut utiliser des constructeurs un peu plus spéciaux :

— Passer par `np.zeros_like(array_like, dtype)` : on crée un tableau dont la forme est basée sur celle de `array_like` dont les objets seront du type `dtype`. Pour le reste, c'est identique à `np.zeros`. Exemple :

```
import numpy as np

initial = [[1, 2, 3, 4], [1, 2, 3, 4]]
zeros = np.zeros_like(initial, int)
# On cree un tableau 2x4 (dimensions de la liste) d'entiers
```

— Passer par `np.ones_like(array_like, dtype)` : on crée un tableau dont la forme est basée sur celle de `array_like` dont les objets seront du type `dtype`. Pour le reste, c'est identique à `np.ones`. Exemple :

```
import numpy as np
```

```
initial = ((1, 2, 3), (1, 2, 3), (1, 2, 3))
ones = np.ones_like(initial, complex)
# On cree un tableau 3x3 (dimensions du tuple) de complexes
```

— Passer par `np.empty_like(array_like, dtype)` : on crée un tableau dont la forme est basée sur celle de `array_like` dont les objets seront du type `dtype`. Pour le reste, c'est identique à `np.empty`. Exemple :

```
import numpy as np

initial = np.ones((100, 100), float)
ones = np.empty_like(initial, str)
# On cree un tableau 100x100 (dimensions de l'array) de str
```

— Passer par `np.full_like(array_like, remplissage, dtype)` : on crée un tableau dont la forme est basée sur celle de `array_like` dont les objets seront du type `dtype`. Pour le reste, c'est identique à `np.ones`. Exemple :

```
import numpy as np

initial = 45
ones = np.full_like(initial, 5, float)
# On cree un tableau de dimension 0 (dimensions d'un scalaire) de float
```

Est-ce possible de prendre un objet, par exemple une liste, et d'en faire un *array* ? Bien sûr ! Il faut simplement utiliser `np.array(args)`. L'ordre des arguments est le suivant : `objet` (l'objet qu'on veut transformer), `dtype` (le type du contenu, à faire attention, car peut mener à des problèmes de conversion), `copie` (si on veut copier l'objet initial dans l'*array*. Très utile pour des objets composés). Les autres arguments sont moins utiles, nous n'en parlerons pas. Voici un exemple :

```
import numpy as np

liste = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
tuples = (("hello", "world"), ("HELLO" "WORLD"))
ones = np.ones((1000, 100), float)

liste_array = np.array(liste, int)
tuples_array = np.array(tuples, str)
ones_array = np.array(ones, float)
```

Vous verrez au cours de votre parcours que les *arrays* de *NumPy* sont très importants.

Avant d'aller plus loin, il est très important de connaître ce que signifie le mot axe dans le contexte d'un *array* avec *NumPy*. Ce mot sera souvent utilisé dans le contexte des sous-sections suivantes et est **TRÈS** souvent utilisé dans le module. C'est un concept assez simple, mais dont les mots pour l'expliquer sont difficiles à trouver. Une image vaut 1000 mots !

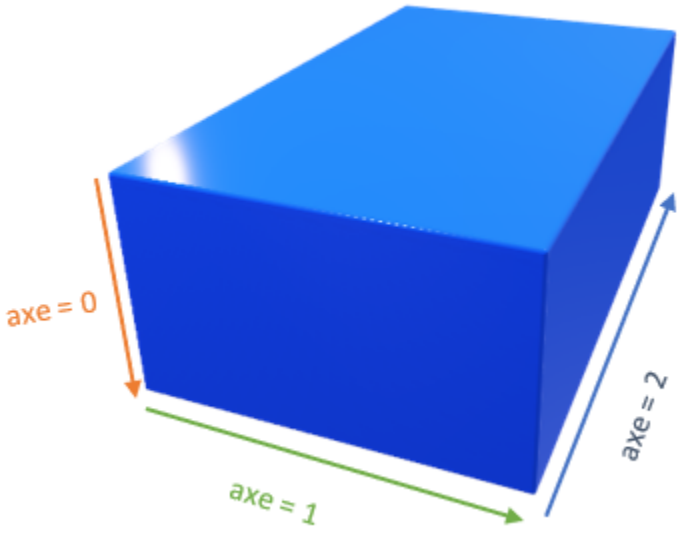


FIGURE 13.1: Représentation graphique d'un *array* et de ses axes

13.1.3 Opérations élémentaires

Comme les tableaux que génère *NumPy* sont très proches des matrices [18], il serait normal que l'on puisse faire des opérations mathématiques. Dans cette sous-section, il sera question des opérations de bases et de quelques particularités. Voici une liste assez exhaustive des opérations possibles, directement appliquées sur des *arrays* :

— Addition : Il est possible d'additionner deux *arrays* ensemble, comme l'addition matricielle. Il faut bien sûr que les deux *arrays* soient de mêmes dimensions. On utilise le symbole usuel de l'addition de types de bases, soit +. Exemple :

```
import numpy as np

a1 = np.ones((4,4), int)
a2 = np.ones((4,4), int)

print(a1 + a2)
```

```
[[2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]]
```

On remarque que le type original des deux *arrays* a été conservé. Si les deux types sont différents, on converti vers le plus général (par exemple, de **int** vers **float**).

— Soustraction : Même principe que l'addition, sauf qu'on utilise le symbole −.

18. En fait, il existe un objet appelé **matrix** dans la librairie de *NumPy*, mais il n'est plus recommandé de l'utiliser et il est possible qu'il disparaisse dans une version future.

— Multiplication terme à terme : Il est possible d'appliquer la multiplication terme à terme, c'est à dire le n-ème de l'*array* de gauche est multiplié par le n-ème terme de l'*array* de droit. Cela est vraiment différent du produit matriciel, vectoriel ou scalaire. On utilise le symbole $*$. Encore une fois, on convertit les types si besoin. Exemple :

```
import numpy as np

a1 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]], int)
a2 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]], int)
print(a1 * a2)
```

```
[[ 1  4  9 16]
 [ 1  4  9 16]]
```

— Division terme à terme : Même principe que la multiplication terme à terme, sauf qu'on utilise le symbole `/` et que, par défaut, on convertie vers un type permettant les décimales.

— Division entière terme à terme : Même principe que la division terme à terme, sauf qu'on utilise `//` et que les décimales sont tronquées.

— Modulo terme à terme : Il est possible d'évaluer le modulo terme à terme. Ce qu'on entend par modulo terme à terme, c'est d'évaluer le n-ème terme de l'*array* de gauche modulo le n-ème terme de l'*array* de droite. On utilise le symbole `%`. Si un des *arrays* est de type **`complex`**, comme le modulo n'est pas défini pour ce type, des erreurs surviendront. Exemple :

```
import numpy as np

a1 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]], int)
a2 = np.array([[1, 3, 4, 5], [1, 3, 4, 5]], int)
print(a1 % a2)
```

```
[[0 2 3 4]
 [0 2 3 4]]
```

— Comparaison terme à terme : Il est aussi possible de comparer le n-ème terme de l'*array* de gauche au n-ème terme de l*array* de droite, et ce avec les opérandes habituels : >, >=, <, <=, =!, ==. L'*array* résultant est de type **`bool`**. À noter que comme le terme **`not`** s'applique sur un booléen (en effet, lorsqu'on fait **`not`** `2 != 3`, on fait **`not`** `True`, ce qui donne `False`) et comme un *array* de booléens n'est pas un booléen, appliquer **`not`** sur un `array` est ambiguë. Nous verrons un peu plus bas comment effectuer la négation sur les éléments d'un *array*. Exemple :

```
import numpy as np

a1 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]], int)
a2 = np.array([[1, 3, 4, 5], [1, 3, 4, 5]], int)
print(a1 == a2)
```

```
[[ True False False False]
 [ True False False False]]
```

— Exposant : Il est possible d'élever le n-ème terme de l'*array* de gauche à une puissance égale au n-ème terme de l'*array* de droite. L'*array* résultant conserve le type, ou convertie au besoin. Le symbole est $**$. Exemple :

```
import numpy as np

a1 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]], int)
a2 = np.array([[1, 3, 4, 5], [1, 3, 4, 5]], int)
print(a1 ** a2)
```

```
[[   1    8   81 1024]
 [   1    8   81 1024]]
```

Les principales opérations de bases ont aussi leur propre méthode. Par exemple, au lieu d'écrire `a1 % a2`, on peut écrire `np.mod(a1, a2)`. De plus, une méthode pour la racine carrée existe, ainsi que des méthodes pour les fonctions trigonométriques, hyperboliques, valeur absolue, etc.

Il est aussi important de savoir que les opérations utilisées ci-dessus peuvent aussi être utilisées lorsqu'un opérande est un scalaire. Par exemple, on peut faire :
`np.array([1, 2, 3]) + 2` *#= np.array([3, 4, 5]*. C'est à la base de ce qu'on appelle la vectorisation du code. Grâce à la manière dont est conçu *NumPy*, ses opérations sont très optimisées et il est possible de rendre un code presque sans boucle, simplement en utilisations les diverses opérations et méthodes que fournit le module. Nous verrons au cours des prochaines sous-sections des étapes qui peuvent mener à un code sans boucles très optimal.

13.1.4 Opérations matricielles et vectorielles

Nous avons vu à la sous-section précédente qu'il existe un assez grande quantité d'opérations de bases, considérés comme terme à terme. Mais ce n'est pas tout, *NumPy* ajoute aussi son lot d'opérations matricielles et vectorielles. Premièrement, les opérations entre deux *arrays* :

— Produit matriciel : Il s'agit de l'équivalent du produit matriciel tel qu'on le connaît, $A \times B$. On peut utiliser deux méthodes ainsi qu'un opérateur pour y arriver. La première méthode est `np.matmul(array1, array2)`. Il y a aussi un opérateur équivalent, soit `@` (oui, oui, le «a commercial» / l'arobase !). On peut aussi utiliser une seconde méthode, `np.dot(array1, array2)`, mais on préfère utiliser la première ou son opérateur. Exemple :

```
import numpy as np

a1 = np.ones((5,5), int)
a2 = a1
print(np.matmul(a1, a2))
```

```
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
```

— Produit interne (vecteur) : Si les *arrays* qu'on manipule sont de vecteurs (donc des *arrays* en 1D), on peut utiliser le produit interne (produit scalaire). Comme le produit scalaire est *dot product* en anglais, il serait normal d'utiliser `np.dot(vector1, vector2)`. Eh bien oui, on peut l'utiliser, mais ce n'est pas la seule option ! En effet, on peut utiliser les mêmes méthodes qu'au point précédent, soit `matmul` et `@`. De plus, on peut utiliser la méthode `np.inner(vector1, vector2)`. Ça en fait pas mal à se souvenir. Exemple :

```
import nunpy as np

a1 = np.array([1, 2, 3, 4])
a2 = a1
print(np.inner(a1, a2))
print(np.inner(a1, a2) == np.dot(a1, a2) and a1 @ a2 == np.dot(a1, a2))
```

```
30
True
```

Il faut faire très attention aux dimensions des tableaux avec lesquels on travaille, car tout dépend de ceux-ci lorsqu'on utilise certaines méthodes. On voit très bien cela, car `np.dot` ou encore `np.matmul` donne des résultats différents lorsque les dimensions sont différentes. À noter qu'il existe aussi `np.vdot` qui fait la même chose que `np.dot`, mais si le premier *array* est complexe, il prend le conjugué complexe du second *array*.

— Produit vectoriel : Il peut être aussi intéressant de calculer le produit vectoriel lorsqu'on est dans l'espace $\mathbb{R}^3$. Pour ce faire, on utilise la méthode `np.cross(vector1, vector2)`. Lorsque les deux *arrays* contiennent trois composantes, il exécute le produit vectoriel, sinon il exécute autre chose dont nous ne parlerons pas ici. Vous pouvez toujours aller lire la documentation pour savoir ce qui se passe. Exemple :

```
import numpy as np

a1 = np.array([1, 2, 3])
a2 = a1
print(np.cross(a1, a2))
a2 = np.array([3, 2, 1])
print(np.cross(a1, a2))
```

```
[0 0 0]
[-5 10 -5]
```

Avant de conclure cette partie sur les opérations matricielles sur deux *arrays*, il est important de savoir que la fonction `np.dot` est aussi disponible en méthode de classe de `ndarray`. Par exemple, si `a1` et `a2` sont deux *arrays*, on peut écrire `a1.dot(a2)`. Finalement, dernier point avant de passer à un autre sujet à propos des opérations matricielles, il est possible d'utiliser une méthode assez générale qui généralise en quelque sorte le produit matriciel et le produit interne. Il s'agit de `np.tensordot(array1, array2, axe)`. L'ajout de l'argument `axe` permet de passer justement à calcul différent. Si cet argument vaut 0, on fait le produit tensoriel. Si cet argument vaut 1 (pour *arrays* en 2D et plus), on fait le produit scalaire tensoriel. Finalement, si cet argument vaut 2 (pour *arrays* en 3D et plus), on fait la double contraction tensorielle. Selon les besoins, *NumPy*

offre plusieurs possibilités.

Parlons maintenant des opérations matricielles qui s'appliquent sur seulement un *array*. Nous n'entrerons pas vraiment dans l'algèbre linéaire sous-jacente , mais nous présenterons les différentes méthodes qui permettent de faire de l'algèbre linéaire assez poussée [19].

Ref vers alg. lin.

Premièrement, une opération assez simple, mais très importante dans plusieurs calculs : la transposition. Il existe deux moyens pour y arriver. On peut d'une part passer par la fonction `transpose` de *NumPy*. Cette fonction prend deux arguments. Le premier est l'*array* en question, alors que le second (optionnel) prend une liste d'entier. Cette liste doit contenir le nouvel ordre des axes. Par défaut, le module renverse l'ordre. Voici un exemple d'utilisation :

```
import numpy as np

array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
transposition = np.transpose(array)
transpotion_avec_arg_optionnel = np.transpose(array, [0, 1])
# [0, 1] signifie qu'on remplace les lignes par les colonnes
```

À noter que les deux *arrays* résultant de la transposition sont équivalents. Par contre dans le cas d'un *array* en $3D$, on pourrait effectuer toutes sortes de permutations d'axes :

```
array = np.array([[[57, 57, 59],
         [ 9, 51, 85],
         [69, 77, 94]],

        [[24,  5, 11],
         [73,  7, 88],
         [64, 48,  1]],

        [[66, 61, 46],
         [ 5, 45,  1],
         [43, 64, 30]]])
transpose_perm1 = np.transpose(array, [0, 2, 1])
transpose_perm2 = np.transpose(array, [2, 1, 0])
transpose_perm3 = np.transpose(array, [1, 2, 0])
```

La deuxième manière de transposer un *array* est d'utiliser la méthode directement de la classe `ndarray`. On peut utiliser la propriété `T` ou bien la méthode `transpose()`. Cette dernière s'utilise comme la première manière, à l'exception qu'on ne lui passe pas d'*array* en paramètre (on prend celui courant). Dans le cas de la propriété, on l'appelle sans parenthèse et est assez de base comparativement à la méthode. En effet, on ne fait que le cas par défaut, soit inverser l'ordre des axes. Voici des exemples d'utilisation :

```
import numpy as np
```

19. Il ne sera pas question d'une présentation exhaustive, mais plus d'une présentation de ce qui est le plus pertinent en physique.

```
array = np.array([[1, 2, 3], [1, 2, 3]])
transpose_propriete = array.T
transpose_methode = array.transpose()
```

À noter que ce code effectue la même chose, car on ne fournit pas d'arguments à la méthode.

Deuxièmement, il pourrait être pertinent de calculer la trace d'une matrice. Cela est faisable de deux manière totalement équivalentes, mais d'arguments (légèrement) différents. On peut passer par la méthode statique ou celle qui s'applique directement sur un objet `ndarray`. Dans le premier cas, les arguments qui peuvent être pertinents sont : l'*array*, un *offset* si on veut calculer une autre diagonale que celle principale (facultatif, défaut à 0) et le numéro d'un premier axe ainsi que d'un second axe pour le cas des *arrays* à dimensions supérieures à 2. Ces deux derniers arguments sont facultatifs et spécifient un *sous-array* en $2D$ dont on calculera la trace. La seconde méthode est tout simplement la même, sauf qu'on omet de spécifier un *array*. Voici un exemple d'utilisation :

```
import numpy as np

array = np.ones((3,3))
print(array.trace())
```

Le résultat est :

```
3
```

Troisièmement, on peut facilement calculer le conjugué complexe d'un *array* à l'aide de `conj` ou `conjugate`. Comme ce qui a été présenté précédemment, ce deux méthodes peuvent prendre un *array* en argument ou bien être utilisées directement sur un *array*. Voici un exemple d'utilisation :

```
import numpy as np

array = np.ones((3,3)) + 2j
conj = array.conj()
conj_2 = array.conjugate()
conj_3 = np.conj(array)
conj_4 = np.conjugate(array)
```

Ces méthodes sont toutes équivalentes.

Finalement, on parlera du déterminant d'une matrice. Dans le cas d'un *array NumPy*, il faudra faire appel au package `linalg` de *NumPy*. Ensuite, on peut directement utiliser la méthode `det` qui prend en argument l'*array* dont on veut calculer le déterminant. Voici un exemple d'utilisation :

```
import numpy as np

array = np.array([[-1, 2, 3], [4, 5, 6], [7, 8, 9]])
det = np.linalg.det(array)
print(det)
```

On obtient le résultat :

```
6.000000000000016
```

À noter que la vraie valeur est 6, mais c'est sûrement à cause d'un problème de mémoire du point flottant qu'on obtient une légère différence.

13.1.5 Fonctions et méthodes intéressantes

En plus des fonctions pour les opérations mathématiques, il existe des fonctions intéressantes. Par exemple, on peut convertir des radians en degrés avec `np.rad2deg(angle)`. Comme il a été mentionné plus haut, on ne peut inverser des booléens avec **not** lorsque ceux-ci sont dans un *array*. Pour se faire, il existe quelques possibilités. Par exemple, on peut utiliser la méthode `np.invert` qui inverse les valeurs d'un *array*. Attention, cette méthode ne marche que pour les types **bool** et les entiers, mais de préférence à utiliser seulement pour inverser des `True` en `False` (ou l'inverse). On peut aussi utiliser l'opérateur non bit-à-bit de deux manières possibles : en utilisant le symbole, par exemple `~(np.ones(5) > 0)` *#= np.array([False, False, False, False, False])*, ou encore en utilisant la fonction associée, soit `np.bitwise_not`, qui fait exactement la même chose.

Il a été mentionné que l'utilisation de **not** est ambiguë dans le cas des *arrays*, utiliser des *arrays* dans des conditions peut la plupart du temps aussi être ambiguë. Dans le cas où l'on veut vraiment vérifier si un *array* contient une certaine valeur booléenne, on peut passer par la méthode **any**`(array)` de *NumPy*[20]. Dans le cas où l'on veut vérifier si tous les éléments d'un *array* sont une certaine valeur booléenne, on peut passer par la méthode **all**`(array)`, encore une fois de *NumPy*[21]. Le lecteur doit comprendre la différence fondamentale entre les deux méthodes. La première vérifie si seulement un seul élément de l'*array* est `True`, alors que la seconde vérifie si tous les éléments sont `True`. À noter qu'il existe une version statique (donc prenant un *array* en argument) ainsi qu'une version non statique (donc prenant l'*array* courant comme référence). Voici quelques exemples d'utilisation :

```
import numpy as np

all_true = np.array([True, True, True], dtype=bool)
not_all_true = np.array([True, False, True], dtype=bool)
array = np.ones((3,))
autre_array = np.array([0, 1, 2])

if all_true.all():
    print("Toutes les valeurs sont True")

if not_all_true.any():
    print("Au moins une valeur est True")

if np.all(array > 0):
```

20. Bien que la même méthode existe *builtin* dans le langage *Python*, la version de *NumPy* est préférable.
21. Comme pour la méthode **any**`()`, il existe une variante déjà présente à la base. Par contre, on préfère utiliser la version de *NumPy*.

```
    print("Toutes les valeurs sont superieures a 2")

if np.any(autre_array == 0):
    print("Au moins une valeur nulle")
```

On aurait la sortie suivante :

```
Toutes les valeurs sont True
Au moins une valeur est True
Toutes les valeurs sont superieures a 2
Au moins une valeur nulle
```

Bien souvent, il se peut que l'on soit obligé de comparer deux *arrays*. Il serait donc tentant d'utiliser l'opérateur de comparaison d'égalité, ==. Or, comme on l'a mentionné plus haut, cela effectue une comparaison terme à terme donc on n'obtient pas de valeur booléenne. Pour en obtenir une, on peut procéder deux manières : utiliser **all** ou bien `array_equal` de *NumPy*. Dans les deux cas, on devrait obtenir le même résultat. Voici un exemple :

```
import numpy as np

a1 = np.ones((2,2))
a2 = np.ones((2,3))
print((a1 == a1).all() == np.array_equal(a1, a1))
print(np.array_equal(a1, a2))
```

Et on a comme sortie :

```
True
False
```

On remarque que, comme `a1` et `a2` ont une forme différente, ils ne peuvent être égaux.

Maintenant que l'aspect booléen des *arrays* est passé, entrons dans la mécanique un peu plus interne des *arrays* avec quelques méthodes qui peuvent sembler inutiles à première vue, mais qui peuvent s'avérer fondamentales et importantes dans des projets d'envergure. D'abord, présentons quelques méthodes[22] permettant d'avoir des *arrays* ayant quelques propriétés alléchantes. On a déjà vu plus tôt qu'il existe la fonction **range** qui permet de construire une intervalle. Avec *NumPy*, il existe une méthode équivalente, mais plus générale, car elle permet d'avoir des bornes et un pas réels (pas juste entiers). Cette méthode s'appelle `arange`. Voici un exemple d'utilisation :

```
import numpy as np

intervalle = np.arange(0.5, 10.5, 0.75)
print(intervalle)
```

22. Il s'agit en quelque sorte de constructeurs, car on obtient un *array* en fin de compte.

On obtient :

```
[ 0.5   1.25  2.    2.75  3.5   4.25  5.    5.75  6.5   7.25  8. 8.75
9.5  10.25]
```

Il existe aussi une autre méthode un but semblable, mais procédant différemment, `linspace`. En effet, cette méthode prend un borne inférieure et une supérieure, mais au lieu de prendre un pas, elle prend un nombre de points. De plus, contrairement à **`range`** et `arange`, `linspace` inclue la borne supérieure. Voici un exemple d'utilisation :

```
import numpy as np

intervalle = np.linspace(0, 10, 10)
print(intervalle)
print(len(intervalle))
```

On a comme sortie :

```
[ 0.          1.11111111  2.22222222  3.33333333  4.44444444
5.55555556  6.66666667  7.77777778  8.88888889 10.        ]
10
```

Il pourrait être intéressant, voire nécessaire, d'avoir ce type d'*array* de valeurs contiguës, mais en plusieures dimensions. Pour se faire, il faut utiliser la méthode `reshape`. Cette méthode prend comme argument la nouvelle forme que l'on désire. Par exemple, si je veux une matrice ayant comme valeur les entiers de l'ensemble $[0, 25)$, on peut faire :

```
import numpy as np

array = np.arange(0, 25, 1).reshape(5, 5)
# Equivalent:
array = np.arange(0, 25, 1)
array = np.reshape(array, (5, 5))
print(array)
```

On a la matrice :

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

On peut revenir à la configuration initiale de l'*array* à l'aide de la méthode `ravel`. Cette méthode prend un *array* et le ramène en 1D. En reprenant le code précédent :

```
print(array.ravel())
```

La sortie est :

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

On pourrait bien sûr faire `array.reshape(25)`, mais dans certains cas où l'on a plusieurs dimensions et qu'il n'est pas super évident de déterminer combien on a d'éléments, il peut être plus judicieux d'utiliser `ravel`. Finalement, il pourrait parfois être pertinent d'imposer des valeurs maximales et minimales dans un *array*. Si on prend l'exemple précédent, on pourrait être contraint à imposer un maximum à 20. On ne peut avoir des valeurs supérieures. De plus, si on dit que le minimum est à 5, on ne peut avoir des valeurs inférieures. Voici ce qu'on pourrait faire :

```
array = array.clip(5, 20)
print(array)
```

On aurait :

```
[[ 5  5  5  5  5]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 20 20 20 20]]
```

Ce genre de méthode paraît sûrement inutile à première vue, mais dans certains contextes, cela peut-être très important. Par exemple, dans un logiciel de traitement d'image, il pourrait être dangereux d'avoir des pixels d'intensité «négative», donc on pourrait imposer que le minimum soit 0. Ainsi, il ne pourrait y avoir de valeurs négatives.

13.1.6 Nouveaux types et utilité

Comme mentionné à quelques reprises, les types en *Python* n'ont pas une très grande importance pour un programmeur du dimanche. Par contre, ils peuvent avoir une importance cruciale dans certains cas. Avant d'entrer dans le pourquoi du comment les types peuvent être importants, la section 3 pourrait être utile pour se rafraîchir la mémoire à propos des types de bases natifs au langage, car ce qui s'en vient est en quelque sorte la suite. Les types de bases de *Python* sont pour la plupart du temps suffisants pour arriver à nos fins. Les `float`, les `int` et les `complex` peuvent abriter une très grande plage de valeurs. Par contre cela n'est pas «suffisant» dans certains cas. C'est pourquoi *NumPy* implémente en quelque sorte des sous-groupes. En voici quelques uns :

- Entiers signés :
  - `np.int8`
  - `np.int16`
  - `np.int32`
  - `np.int64`
- Entiers non-signés :
  - `np.uint8`
  - `np.uint16`
  - `np.uint32`
  - `np.uint64`
- Réels :

    - `np.float32`
    - `np.float64`
- Complexes :
    - `np.complex64`
    - `np.complex128`

Lorsqu'on utilise des *arrays*, il est très important de savoir quel est le type des éléments qui les composeront. Il n'est pas très rare de voir dans certains cas des problèmes de conversion lorsqu'on gère des tableaux dans certains contextes. Par exemple, additionner des images qui ne sont pas encodés sur le même nombre de bits peut s'avérer un cauchemar si on ne sait pas comment gérer les différents types. Dans des cas de travaux scolaires, les tableaux utilisés seront certainement de petites tailles (quelques centaines d'éléments), donc le type ne sera pas super pertinent. Par contre, si on travaille sur un projet d'envergure, avec plusieurs milliers (voire millions) d'entrées [23] et si nos ressources sont limitées, on ne peut peut-être pas se permettre de tout encoder en 64 bits. Si on sait que les entrées sont toutes entières, mais allant seulement dans un intervalle de -100 à 100, on peut facilement tout encoder dans seulement 8 bits (utiliser `np.int8`). Il se peut donc que le type de données qu'on utilise devienne un sujet très sérieux. Heureusement, *NumPy* est là !

23. Une image 2465x855 contient 2107575 éléments. Cette quantité est relativement petite pour les machines actuelles, mais il vaut mieux prendre le moins de mémoire possible.

13.2 SciPy

Le module *SciPy* est en quelque sorte le fils de *NumPy*, car non seulement il en reprend certaines méthodes, mais il en ajoute des nouvelles s'utilisant très bien avec *NumPy* et `ndarray`. Ce module est à utiliser lorsqu'on veut utiliser des routines mathématiques. Que l'on veuille utiliser des constantes physiques, des fonctions spéciales, des méthodes d'intégrations numériques ou bien des fonctions servant à la manipulation d'images, *SciPy* est la principale référence à utiliser.

13.2.1 Installation

Voir 13.1.1, car l'installation de *SciPy* se fait majoritairement de la même manière que *NumPy*.

13.2.2 Les premiers pas

Comme pour *NumPy*, l'utilisation de *SciPy* se fait assez facilement. Le *namesapce* principal de *SciPy*, qu'on obtient en faisant `import scipy` contient principalement des méthodes déjà existantes dans *NumPy* et il n'est pas très pertinent d'utiliser cette syntaxe. Par contre, *SciPy* est composé de quelques sous-modules indépendants. Voici quelques exemples de sous-modules :

- `scipy.constants` contient des constantes mathématiques et physiques
- `scipy.integrate` contient des routines d'intégration numérique
- `scipy.linalg` contient des algorithmes d'algèbre linéaire (basé sur `np.linalg`, mais des fonctions de même nom peuvent avoir des fonctionnalités différentes)
- `scipy.ndimage` contient des fonctions pour le traitement d'image
- `scipy.optimize` contient des fonctions pour l'optimisation

On importe les sous-modules dont on a besoin. Par exemple :

```
from scipy import optimize, constants
```

L'intégration de *SciPy* dans son code se fait très bien. Il ne faut pas avoir peur de l'utiliser, même si la documentation peut paraître compliquée à comprendre. Étant donné que *SciPy* est beaucoup plus spécifique que *NumPy*, une introduction générale est difficile à faire. Des sections ultérieures traiterons plus en détails le fonctionnement de *SciPy* dans des contextes spécifiques, comme l'algèbre linéaire, la résolution d'équations différentielles, etc.

13.3 Pandas

Le module *pandas* est un module couramment utilisé pour la lecture et l'écriture de fichiers de données. Il se base sur un objet *maison*, le `DataFrame` qui permet une lecture rapide et efficace. Le principal avantage de *pandas* contrairement à d'autres modules pour lire des fichiers est qu'il est extrêmement polyvalent. On peut lire plusieurs types de fichiers, autant des fichiers *csv* que des fichiers *html* ou encore *HDF5*. La représentation qu'il donne des données est aussi très intuitive : si possible, il extrait les noms des colonnes et des lignes à partir des données et ceux-ci peuvent être utilisés comme index. Au lieu d'utiliser des chiffres comme ce qu'on doit faire avec les `ndarray`, on peut simplement faire référence aux colonnes avec leur nom. Ainsi, si une série de données contient la colonne "naissance", on peut simplement y accéder directement avec le nom, sans nécessairement avoir besoin de savoir quel est son index associé (i.e. s'agit-il de la première colonne, de la seconde ou de la dernière ?). Il est aussi possible d'appliquer des transformations sur les données ainsi que sur la structure même du `DataFrame`, comme combiner plusieurs `DataFrame`. Bref, ce module est très polyvalent, en plus d'être assez compatible avec d'autres comme *NumPy*.

13.3.1 Installation

L'installation de *pandas* se fait assez facilement par `pip` ou, si on utilise *Anaconda*, `conda`. Comme pour *NumPy*, *pandas* vient directement avec l'interpréteur *Anaconda*. Par contre, s'il n'est pas installé, `pip` et compagnie sont toujours disponibles. Pour plus d'information, visitez :
`https://pandas.pydata.org/getting_started.html`

13.3.2 Les premiers pas

L'utilisation de *pandas* se veut simple. On importe le module dans son code, ici nous ferons `import pandas as pd`. La racine du module contient des méthodes statiques ainsi que des constructeurs d'objets, comme le fameux `DataFrame`. Un survol des méthodes statiques les plus pertinentes se feront un peu plus loin. Pour l'instant, on se concentre sur le morceau le plus important de *pandas*, le `DataFrame`.

La création d'un tel objet est assez simple, on procède comme suit [24] :

```
import pandas as pd

df = pd.DataFrame(data, index, columns, dtype, copy)
```

Voyons maintenant les paramètre plus en détail :

1. `data` : structure de données. Peut être pas mal n'importe quoi qui est considéré *array-like*, soit un tuple, une liste ou bien sûr un `ndarray`. Il peut aussi être un dictionnaire ou un autre `DataFrame`. Cet argument est *optionnel*, mais un `DataFrame` sans données est vraiment inutile... ATTENTION : les données passées en argument doivent être en deux dimensions (au maximum).

24. Construire un `DataFrame` à partir d'un fichier de données sera vu plus loin.

2. `index` (optionnel) : structure de données. Encore une fois, peut être pas mal n'importe quoi qui est considéré *array-like*. Sert à *nommer* les lignes du `DataFrame`. Par défaut, ce sera une liste numérique allant de 0 à $n-1$ où n est le nombre de lignes.
3. `columns` (optionnel) : structure de données. Très similaire à `index`. Par contre, sert à *nommer* les colonnes. Par défaut, ce sera aussi une liste numérique de 0 à $m-1$ où m est le nombres de colonnes.
4. `dtype` (optionnel) : type des données. On peut mélanger des **`str`** avec des **`int`**, mais ce n'est pas vraiment recommandé. Par défaut, *pandas* infère le type à partir des données.
5. `copy` (optionnel) : booléen. Spécifie si *pandas* doit copier les valeurs de la structure originale. Aucune copie faite par défaut.

Voici quelques exemples de création de `DataFrame` :

```
import pandas as pd
import numpy as np

data_dict = {"Colonne 1": [4.5, 1.67, 98], "Colonne 2": [1.3, 0.09, 4e-7]}
df_dict = pd.DataFrame(data_dict) # Infere le nom des cols avec le dict

data_liste = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
df_liste = pd.DataFrame(data_liste, index=["1 - 5", "6 - 10", "11-15"])

data_array = np.arange(25).reshape(5, 5)
df_array = pd.DataFrame(data_array, columns=["C1", "C2", "C3", "C4", "C5"])

print(df_dict, "\n")
print(df_liste, "\n")
print(df_df)
```

On a en console :

```
   Colonne 1     Colonne 2
0       4.50  1.300000e+00
1       1.67  9.000000e-02
2      98.00  4.000000e-07

          0   1   2   3   4
1 - 5     1   2   3   4   5
6 - 10    6   7   8   9  10
11-15    11  12  13  14  15

   C1  C2  C3  C4  C5
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
```

La classe `DataFrame` offre plusieurs méthodes qui peuvent être intéressantes dans un cadre statistique et d'analyse de données, on peut appliquer des transformations sur les données, on peut calculer la covariance et la corrélation entre les colonnes. De plus, la classe contient des attributs permettant d'accéder à certaines lignes, colonnes ou tout simplement certains éléments du tableau.

13.3.3 Attributs et méthodes de DataFrame

Nous verrons ici les principaux attributs et principales méthodes qu'offre `DataFrame`[25]. Il y aura aussi des exemples d'utilisation à chaque attribut et méthodes. Dans ces exemples, nous considérerons le code suivant (sauf avis contraire) :

```
import pandas as pd
import numpy as np

data = np.arange(9).reshape(3, 3)
cols = ["C1", "C2", "C3"]
index = ["L1", "L2", "L3"]

df = pd.DataFrame(data, index, cols)
```

Pour les attributs :

— `at` : Permet d'accéder à un élément du `DataFrame` par le nom de la ligne et de la colonne. Exemple :

```
print(df.at["L2", "C2"]) # Donne 4
```

— `iat` : Permet d'accéder à un élément du `DataFrame` par l'index numérique de la ligne et de la colonne. Exemple :

```
print(df.iat[1, 1]) # Donne 4
```

— `loc` : Permet d'accéder à plusieurs éléments du `DataFrame` par le nom des lignes et des colonnes. Exemple :

```
print(df.loc["L2"]) # Donne l'info sur la ligne 2
print(df.loc[:, "C1"]) # Donne l'info sur la colonne 1
print(df.loc[["L1", "L3"], ["C1"]]) # Donne l'info sur la ligne 1
# et 3 de la colonne 1
```

— `iloc` : Permet d'accéder à plusieurs éléments du `DataFrame` par l'index numérique des lignes et des colonnes. Exemple :

```
print(df.iloc[1]) # Donne l'info sur la ligne 2
print(df.iloc[:, 0]) # Donne l'info sur la colonne 1
print(df.iloc[[0, 2], [0]]) # Donne l'info sur la ligne 1
# et 3 de la colonne 1
```

25. Cette présentation n'est pas exhaustive. Seulement les aspects les plus intéressants et *utiles* selon les auteurs seront approchés.

— `columns` : Permet d'accéder au nom des colonnes. Exemple :

```
print(df.columns) # Donne Index(['C1', 'C2', 'C3'], dtype='object')
```

— `index` : Permet d'accéder au nom des lignes. Exemple :

```
print(df.index) # Donne Index(['L1', 'L2', 'L3'], dtype='object')
```

— `values` : Retourne une représentation des données dans un `ndarray`. Exemple :

```
print(df.values)
# Donne:
# array([[0, 1, 2],
#        [3, 4, 5],
#        [6, 7, 8]])
```

D'autres attributs sont aussi disponibles, notamment pour obtenir la taille du tableau ou le nombre d'éléments qu'il contient.

Pour les méthodes :

— Méthodes arithmétique :

— `add`

— Continuer avec les autres pertinentes

14 Algèbre linéaire

14.1 Systèmes d'équations

Les systèmes d'équations linéaires jouent un grand rôle en sciences. Ils sont par exemple présents en électronique lorsqu'on travaille dans le domaine de Laplace. Sous sa forme la plus simple, une équation linéaire est de la forme :

$$y = ax$$

et en généralisant à plusieurs dimensions, on aurait quelque chose du genre, sous forme matricielle :

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \equiv \mathbf{a}\vec{x} = \vec{b} \tag{14.1}$$

Résoudre un problème à plusieurs dimensions peut être relativement fastidieux lorsqu'on travaille manuellement, mais lorsqu'on a la chance de résoudre numériquement, le calcul est beaucoup plus rapide, notamment par l'utilisation d'algorithme très rapides et de la puissance de calcul des machines modernes. Tel que mentionné à la section 13.1, *NumPy* implémente plusieurs algorithmes utiles à l'algèbre linéaire, dont une fonction permettant de résoudre un système d'équation linéaire comme (14.1). Pour y arriver, on utilise `solve(a, b)` de `np.linalg`. `a` est simplement la matrice de coefficients (`a` dans l'exemple précédent) et `b` est simplement le vecteur de valeurs des variables indépendantes ($\vec{b}$ dans l'exemple précédent). Considérons l'exemple suivant : on a un circuit électronique, présenté à la figure 14.1. On peut écrire les équations dans chaque boucle de courant (tâche laissée en exercice au lecteur) et, après avoir réarranger les termes ensemble, on a les équations suivantes :

$$\begin{aligned} 76I_1 - 25I_2 - 50I_3 + 0I_4 + 0I_5 + 0I_6 &= 10 \\ -25I_1 + 56I_2 - 1I_3 - 30I_4 + 0I_5 + 0I_6 &= 0 \\ -50I_1 - 1I_2 + 106I_3 - 55I_4 + 0I_5 + 0I_6 &= 0 \\ 0I_1 - 30I_2 - 55I_3 + 160I_4 - 25I_5 - 50I_6 &= 0 \\ 0I_1 + 0I_2 + 0I_3 - 25I_4 + 56I_5 - 1I_6 &= 0 \\ 0I_1 + 0I_2 + 0I_3 - 50I_4 - 1I_5 + 106I_6 &= 0 \end{aligned}$$

On peut écrire sous forme matricielle :

$$\begin{bmatrix} 76 & -25 & -50 & 0 & 0 & 0 \\ -25 & 56 & -1 & -30 & 0 & 0 \\ -50 & -1 & 106 & -55 & 0 & 0 \\ 0 & -30 & -55 & 160 & -25 & -50 \\ 0 & 0 & 0 & -25 & 56 & -1 \\ 0 & 0 & 0 & -50 & -1 & 106 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ I_6 \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

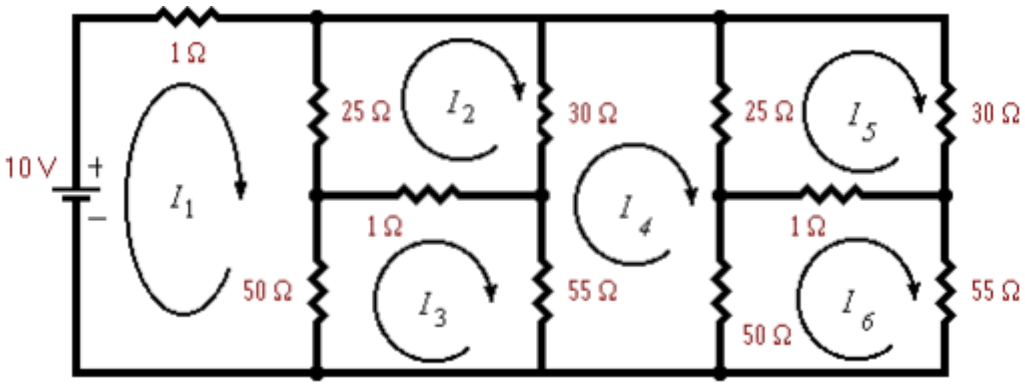


FIGURE 14.1: Exemple de circuit électrique

Écrivons maintenant un programme *Python* permettant de calculer les différentes valeurs de I_k, avec $k \in \{1, 2, 3, 4, 5, 6\}$:

```
import numpy as np

a = [[76, −25, −50, 0, 0, 0], [−25, 56, −1, −30, 0, 0],
     [−50, −1, 106, −55, 0, 0], [0, −30, −55, 160, −25, −50],
     [0, 0, 0, −25, 56, −1], [0, 0, 0, −50, −1, 106]]

a = np.array(a)

b = np.zeros(6)
b[0] = 10

I = np.linalg.solve(a, b)
```

Les valeurs trouvées sont :

```
[0.47815666 0.34784185 0.3528772  0.23907833 0.10876352 0.11379887]
```

On peut vérifier la solution en faisant :

```
print(np.allclose(a @ I, b))
```

Et on a comme sortie :

```
True
```

14.2 Décomposition matricielle

La décomposition matricielle peut s'avérer très utile pour divers calculs. Cela permet notamment de calculer les valeurs et les vecteurs propres plus rapidement (numériquement) en utilisant la décomposition QR. En informatique, les calculs matriciels relativement complexe (inversion, résolution systèmes d'équations, vecteurs et valeurs propres, etc.) sont majoritairement effectués par l'entremise de la décomposition matricielle

14.2.1 Décomposition LU

La décomposition LU est une décomposition qui prend une matrice A et qui la décompose en deux autres matrices L et U de sorte que $A = LU$. La matrice L est une matrice triangulaire inférieure alors que la matrice U est une matrice triangulaire supérieure. La décomposition par seulement l'utilisation de L et U est basique. On peut ajouter une matrice P de permutation de lignes ou de colonnes. Dans le cas de la matrice de permutations, on a $A = PLU$. On peut utiliser cette méthode de décomposition pour résoudre des systèmes d'équation ou lorsqu'on veut inverser une matrice. Naturellement, il existe quelques méthodes pour effectuer la décomposition LU en *Python* grâce au module *SciPy*[26].

Premièrement, il y a `scipy.linalg.lu`. Cette permière routine effectue la décomposition $A = PLU$. Sans se soucier des arguments par défaut, elle prend une matrice et donne la matrice de permutation, la matrice triangulaire inférieure et celle supérieure. Voici un exemple :

```
import numpy as np
from scipy import linalg

a = np.array([[ 8, -9,  0,  9],
       [-7, -2, -8,  1],
       [ 7, -2,  4,  2],
       [ 3, -3,  4, -1]])

P, L, U = linlag.lu(a)

print(P)
print(L)
print(U)
print(P @ L @ U)
```

On obtient :

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]

[[ 1.          0.          0.          0.        ]
 [-0.875       1.          0.          0.        ]
 [ 0.375      -0.03797468  1.          0.        ]
 [ 0.875      -0.59493671 -0.20547945  1.        ]]

[[ 8.         -9.          0.          9.        ]
 [ 0.         -9.875      -8.          8.875     ]
 [ 0.          0.          3.69620253 -4.03797468]
 [ 0.          0.          0.         -1.42465753]]
```

26. Ce ne sont pas tous les arguments des méthodes qui seront présentés. Pour plus d'information, voir la documentation de *SciPy*.

```
[[ 8. −9.  0.  9.]
 [−7. −2. −8.  1.]
 [ 7. −2.  4.  2.]
 [ 3. −3.  4. −1.]]
```

Deuxièmement, on peut utiliser `scipy.linalg.lu_factor`. Elle ressemble à la méthode précédente, mais elle retourne deux objets : la matrice LU qui contient U dans son triangle supérieur et L dans son triangle inférieur, ainsi qu'une liste de pivots. En reprenant la matrice `a` de l'exemple précédent, voici comment l'utiliser :

```
LU, pivots = linalg.lu_factor(a)

print(LU)
print(pivots)
```

On obtient :

```
[[ 8.          −9.          0.          9.        ]
 [−0.875       −9.875      −8.          8.875     ]
 [ 0.375       −0.03797468  3.69620253 −4.03797468]
 [ 0.875       −0.59493671 −0.20547945 −1.42465753]]

[0 1 3 3]
```

La liste de pivots doit être interprétée comme suit : la ligne i a été échangée avec le i^e élément de la liste de pivots. Dans l'exemple précédent, on voit que les deux premières lignes restent inchangées, alors que la troisième ($i = 2$) est échangée avec la dernière ($i = 3$), c'est pourquoi on a `[0, 1, 3, 3]` au lieu de `[0, 1, 2, 3]`, cette-dernière représentant une matrice ne faisant aucune permutation.

Finalement, on peut utiliser un algorithme permettant de résoudre un système d'équations linéaire $\mathbf{A}\vec{x} = \vec{b}$, soit `scipy.linalg.lu_solve`. On a préalablement besoin de la décomposition LU de la matrice $\mathbf{A}$, plus précisément de la combinaison LU en une seule matrice et de la liste de pivots, donc on doit avoir déjà appliqué `scipy.linalg.lu_factor` avant d'utiliser cette routine. On doit aussi donner le vecteur $\vec{b}$ en second argument. Reprenons l'exemple du circuit électrique utilisé dans la sous-section sur les systèmes d'équations linéaires :

```
a = [[76, −25, −50, 0, 0, 0], [−25, 56, −1, −30, 0, 0],
     [−50, −1, 106, −55, 0, 0], [0, −30, −55, 160, −25, −50],
     [0, 0, 0, −25, 56, −1], [0, 0, 0, −50, −1, 106]]

a = np.array(a)

b = np.zeros(6)
b[0] = 10

LU, pivots = linalg.lu_factor(a)
```

```
I = linalg.lu_solve((LU, pivots), b)

print(np.allclose(a @ I, b))
```

On obtient :

```
True
```

Il est important de saisir à quel point la décomposition LU est utile pour le calcul d'inversion de matrice ou celui du déterminant. Pour ce dernier, on peut tout simplement faire :

$$\det\{A\} = \det\{PLU\} = \det\{P\}\det\{L\}\det\{U\} = (-1)^S \left(\prod_{i=0}^{n-1} l_{ii}\right)\left(\prod_{i=0}^{n-1} u_{ii}\right)$$

Ce calcul est excessivement simple, car le déterminant de la matrice P est tout simplement le déterminant de la matrice identité, mais dont on a effectué S permutations. Le déterminant d'une matrice triangulaire, qu'elle soit supérieure ou inférieure, est seulement le produit des éléments de la diagonale. Il est donc très facile de concevoir un algorithme maison permettant de calculer le déterminant d'une matrice :

```
import numpy as np
from scipy import linalg

def determinant(matrice):
    LU, pivots = linalg.lu_factor(matrice)

    # Avec scipy, la matrice L ne contient que des 1 sur la diagonale
    # et ces valeurs ne sont pas prises en compte dans la matrice LU
    # Seule la diagonale de U est presente.
    # Donc prod u_{ii} * prod l_{ii} = prod u_{ii} = prod (lu)_{ii}
    det = np.prod(np.diag(LU))

    # On somme le nombre d'elements tel que pivots[i] != i
    # Cela nous donne le nombre de permutations
    nb_permutations = np.sum(pivots != np.arange(len(pivots)))
    det *= (-1) ** nb_permutations

    return det
```

14.2.2 Décomposition QR

La décomposition QR peut être très intéressante lorsqu'on doit calculer des valeurs et des vecteurs propres. Cette décomposition se base sur le produit d'une matrice orthogonal Q et d'une matrice triangulaire supérieure R tel que la matrice à décomposer, A, peut s'écrire comme $A = QR$. Il est important de savoir que la décomposition QR n'a aucun lien avec les *codes* QR ! Comme il a été mentionné plus haut, la matrice Q est orthonogonale dont les colonnes sont des vecteurs orthonormaux, ce qui implique que $Q^\dagger Q = \mathbb{I}$, où $\mathbb{I}$ est la matrice identitée. Il existe trois grandes

méthodes pour calculer la matrice Q et la matrice R : la méthode de Gram–Schmidt, la méthode de Householder et la méthode de Givens. Chaque méthode a ses avantages et ses inconvénients. *SciPy* implémente la décomposition QR dans le sous-module `linalg`. Pour obtenir les matrices Q et R, on utilise `scipy.linalg.qr` et on fournit la matrice qu'on veut décomposer. Voici un exemple d'utilisation :

```
import numpy as np
from scipy import linalg

a = np.array([[ 8, -9,  0,  9],
        [-7, -2, -8,  1],
        [ 7, -2,  4,  2],
        [ 3, -3,  4, -1]])

Q, R = linalg.qr(a)

print(Q)
print(R)
print(np.allclose(Q @ R, a))
```

La console affiche :

```
[[-0.61177529 -0.67475085 -0.23817627 -0.33721559]
 [ 0.53530338 -0.68838218 -0.10244488  0.47862858]
 [-0.53530338  0.17039163 -0.23203604  0.79408832]
 [-0.22941573 -0.20446995  0.93751607  0.16316883]]

[[-13.07669683   6.19422481  -7.34130348  -5.81186526]
 [  0.          7.7221486    5.37074413  -6.2158866 ]
 [  0.          0.           3.64147919  -3.64761942]
 [  0.          0.           0.          -1.13130391]]

True
```

À partir de la matrice Q et de la matrice R, il est possible de calculer itérativement les valeurs et les vecteurs propres d'un de la matrice $Q \times R$. On verra à la sous-section suivante comment, de même que des routines déjà existantes.

14.3 Vecteurs et valeurs propres

Avant d'entrer dans les routines de *NumPy* et de *SciPy*, voyons comment calculer les valeurs et vecteurs propres d'une matrice à partir des matrices Q et R[27] :

```
import numpy as np
from scipy import linalg

def vecteurs_et_valeurs_propres(matrice, precision=1E-6, iter_max=200):
    # Matrice de vap sur la diag
    valeurs_propres = matrice.copy()
    # Matrice de vep
    vecteurs_propres = np.identity(matrice.shape[1])

    # Masque qui permettra de comparer les elements hors-diag seulement
    # 1 -> True, 0 -> False (on masque avec True)
    mask_diag = vecteurs_propres.copy().astype(bool)

    iter_courant = 1
    # On verifie si les elements hors-diag sont proches de l'erreur
    precision_atteinte = np.allclose(np.abs(np.ma.masked_where(
            diag_mask, matrice_diag)), precision)

    while not precision_atteinte and iter_courant < iter_max:
        Q, R = linalg.qr(valeurs_propres)
        valeurs_propres = R @ Q
        vecteurs_propres = vecteurs_propres @ Q
        precision_atteinte = np.allclose(np.abs(np.ma.masked_where(
            diag_mask, matrice_diag)), precision)
        iter_courant += 1

    return np.diag(valeurs_propres), vecteurs_propres
```

Maintenant, voyons ce que *NumPy* et *SciPy* apportent dans le domaine. Premièrement, voyons ce que *NumPy* a à offrir. Une visite de la documentation nous montre que quatre fonctions existent. Commençons avec `numpy.linalg.eig` et `numpy.linalg.eigh`. La première est plus générale que la seconde, car `numpy.linalg.eigh` est pour les matrice hermitienne (donc $A^\dagger = A$). Dans les deux cas, on obtient un tuple dont le premier élément contient les valeurs propres et le second contient les vecteurs propres normalisés. Voici un exemple d'utilisation :

```
import numpy as np

mat_hermitienne = np.array([[-1, 1 - 1j, 1 + 2j, -1j],
        [1 + 1j, 3, -2, 3 - 2j],
        [1 - 2j, -2, 0, 4],
        [1j, 3 + 2j, 4, 2]])
```

27. Un début d'explication mathématique peut être trouvé sur *Wikipédia* : `https://en.wikipedia.org/wiki/QR_algorithm`

```
mat_quelconque = np.array([[-8,  3, -9, -4],
       [ 0,  7,  1,  0],
       [ 6,  7, -8,  7],
       [ 7, -3, -6, -3]])

valeurs_propres_herm, vecteurs_propres_herm = np.linalg.eigh(mat_hermitienne)
valeurs_propres, vecteurs_propres = np.linalg.eig(mat_quelconque)

print(valeurs_propres_herm)
print(vecteurs_propres_herm)
print(valeurs_propres)
print(vecteurs_propres)
```

On a :

```
[-5.56701126 -1.33617386  4.07621947  6.82696565]

[[ 0.38042122+0.j          0.829968  +0.j         -0.38614224+0.j
   0.13163198+0.j        ]
 [-0.13581339+0.31629292j -0.07439741-0.44407534j -0.13595072-0.48708831j
   0.46278607+0.45701977j]
 [-0.34809327+0.56552804j  0.0524583 +0.00254727j -0.26909153+0.64366182j
  -0.11413831+0.23772338j]
 [ 0.32144779-0.43871817j -0.26221406+0.19209543j -0.24565883+0.21931143j
   0.00368045+0.70005821j]]

[-5.46445035+10.76269955j -5.46445035-10.76269955j
 -8.38976321 +0.j          7.3186639  +0.j        ]

[[ 0.61773826+0.j          0.61773826-0.j         -0.7254959 +0.j
  -0.1000465 +0.j        ]
 [-0.00820897+0.03123729j -0.00820897-0.03123729j  0.01949734+0.j
  -0.88601812+0.j        ]
 [-0.2338773 -0.47770627j -0.2338773 +0.47770627j -0.30005939+0.j
  -0.28234199+0.j        ]
 [ 0.12849069-0.56386573j  0.12849069+0.56386573j  0.61906374+0.j
   0.35390055+0.j        ]]
```

Il est important de noter que la i^e valeur propre a comme vecteur propre la i^e colonne de la matrice de vecteurs propres. Dans le cas où l'on veut seulement les valeurs propres, on peut utiliser deux fonctions : `numpy.linalg.eigvals` pour une matrice générale et `numpy.linalg.eigvalsh` pour une matrice hermitienne. L'utilisation de ces fonctions est essentiellement pareille aux deux autres présentées plus haut, sauf qu'on obtient seulement les valeurs propres.

Dans le cas de *SciPy*, on a des fonctions assez spéciales à première vue. On a des fonctions très générales permettant de résoudre un problème aux valeurs propres. On peut les utiliser pour

résoudre l'équation caractéristique. Pour plus d'informations, allez à `https://docs.scipy.org/doc/scipy/reference/linalg.html#module-scipy.linalg` et lisez la documentation des fonction dans la sous-section *Eigenvalue Problems*.

14.4 Pseudo-inverse

14.4.1 Décomposition en valeurs singulières

Considérons la matrice M de dimension $m \times n$ aux coefficients réels ou complexes. On peut alors décomposer la matrice M selon :

$$M = U\Sigma A^\dagger \tag{14.2}$$

où U est une matrice unitaire $m \times m$, Σ est une matrice $m \times n$ dont les coefficients diagonaux sont réels ou nuls et ceux hors diagonaux sont nuls, alors que $A^\dagger$ est la matrice transposée et conjuguée complexe de la matrice V, matrice unitaire $n \times n$. La décomposition en valeur singulière est très utile dans le calcul de la pseudo-inverse. De plus, on a un «semblant» de valeurs et de vecteurs propres avec les valeurs singulières. En effet, on peut écrire :

$$M^\dagger \vec{u} = \sigma \vec{v} \qquad\qquad M\vec{v} = \sigma \vec{u}$$

où σ est la valeur singulière, $\vec{u}$ est le vecteur singulier à gauche pour σ et $\vec{v}$ est le vecteur singulier à droite pour σ. L'équation (14.2) nous renseigne sur les valeurs singulières, car elles correspondent aux coefficient diagonaux. De plus, les colonnes de U correspondent au vecteurs singuliers à gauche et les colonnes de V représentent les vecteurs singuliers à droite. On peut effectuer la décomposition en valeurs singulières à l'aide de *NumPy*. On peut utiliser `numpy.linalg.svd`. Cette fonction prend une matrice en paramètre et retourne les trois matrices de (14.2). Un problème possible qui peut survenir est que cette fonction ne retourne pas la matrice S en tant que tel, mais plutôt une liste des valeurs singulières, donc la diagonale de S. Passons à un exemple d'utilisation :

```
import numpy as np

a = np.array([[1, 0, 0, 0, 2],
        [0, 0, 3, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 4, 0, 0, 0]])

u, s, v_herm = np.linalg.svd(a)

print(u)
print(s)
print(v)
```

On obtient alors :

```
[[ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  0. -1.]
```

```
 [ 1.  0.  0.  0.]]

[4.         3.         2.23606798 0.        ]

[[-0.          1.          0.         -0.          0.        ]
 [-0.          0.          1.         -0.          0.        ]
 [ 0.4472136   0.          0.          0.          0.89442719]
 [ 0.          0.          0.          1.          0.        ]
 [-0.89442719  0.          0.          0.          0.4472136 ]]
```

On peut aussi utiliser *SciPy*. Premièrement, il existe `scipy.linalg.svd`, qui est essentiellement la même chose que la version de *NumPy*. On peut aussi utiliser `scipy.linalg.svdvals` qui prend une matrice et qui retourne une liste des valeurs singulières, en ordre décroissant. Si on a seulement besoin de ces valeurs, il est peut-être mieux d'utiliser cette méthode, et non une autre qui retourne les matrices U et $V^\dagger$ en plus. Finalement, comme on l'a vu, les fonctions ne retournes pas la matrice Σ, mais plutôt les éléments de sa diagonale. Heureusement, il existe `scipy.linalg.diagsvd` qui prend en argument, dans l'ordre, les valeurs singulières (sous forme d'une liste, d'un *array* en 1D), la dimension m et la dimension n. Par exemple, si on a une matrice 4×3 dont les valeurs singulières sont 12.6127658, 11.32625633 et 3.82544851, on donnerait ces valeurs en premier argument, 4 en second argument et 3 en troisième argument. Voici un exemple d'utilisation :

```
import numpy as np
from scipy import linalg

a = np.array([[-10,   7,  -2],
       [  2,   1,   3],
       [ -1,   3,   6],
       [ -2,  -2,   9]])

s = linalg.svdvals(a)

M, N = a.shape

S = linalg.diagsvd(s, M, N)
print(s)
print(S)
```

On a alors :

```
[12.6127658  11.32625633  3.82544851]

[[16.21099102  0.          0.        ]
 [ 0.         11.49574824  0.        ]
 [ 0.          0.          3.0085782 ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]]
```

14.5 Pseudo inverse à partir de la décomposition en valeurs singulières (ou pas...)

On peut utiliser la décomposition en valeurs singulière pour calculer la pseudo inverse d'une matrice quelconque. En effet, la pseudo inverse s'écrit comme :

$$M^+ = V\Sigma^+U^\dagger$$

où M^+ est la pseudo inverse de M et Σ^+ est la pseudo inverse de la matrice Σ. Pour obtenir Σ^+, on remplace tous les coefficients de Σ par leur inverse multiplicatif et on la transpose. Si on ne veut pas utiliser de module pour calculer la pseudo inverse, on pourrait faire :

```
import numpy as np
from scipy import linalg

def pseudo_inverse(matrice):
    M, N = matrice.shape
    U, s, V_h = linalg.svd(matrice)

    # On inverse les valeurs de s qui sont non nulles
    # L'argument 'out' sert a specifier la dimension de l'output
    s_pseudo_inverse = np.divide(1, s, where=s!=0, out=np.zeros_like(s))

    S_pseudo_inverse = linalg.diagsvd(s, M, N).T

    U_h = U.conj().T

    V = V_h.conj().T

    return V @ S_pseudo_inverse @ U_h

a = np.array([[1, 0, 0, 0, 2],
        [0, 0, 3, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 4, 0, 0, 0]])

a_pseudo_inverse = pseudo_inverse(a)

print(a_pseudo_inverse)
```

On a alors :

```
[[0.2        0.         0.         0.        ]
 [0.         0.         0.         0.25      ]
 [0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.4        0.         0.         0.        ]]
```

Maintenant qu'on a vu un algorithme maison permettant de calculer la pseudo inverse d'une matrice, voyons comment on pourrait utiliser *NumPy* et *SciPy* pour résoudre le même problème sans trop prendre de temps. Commençons par *NumPy*. On peut utiliser `numpy.linalg.pinv` qui prend 3 arguments : la matrice à inverser, un paramètre de tolérance sur les valeurs singulières et un booléen pour l'hermiticité de la matrice. Cette fonction utilise la décomposition en valeurs singulières. On l'utilise comme suit :

```
import numpy as np

a = np.array([[1, 0, 0, 0, 2],
        [0, 0, 3, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 4, 0, 0, 0]])

pinv = np.linalg.pinv(a)

print(pinv)
```

On a alors :

```
[[0.2        0.         0.         0.        ]
 [0.         0.         0.         0.25      ]
 [0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.4        0.         0.         0.        ]]
```

SciPy offre un peu plus de diversité. En effet, on compte trois fonctions différentes : `scipy.linalg.pinv`, `scipy.linalg.pinv2` et `scipy.linalg.pinvh`. Les deux premières sont générales, alors que la dernière est pour les matrices hermitiennes. La différence entre `scipy.linalg.pinv` et `scipy.linalg.pinv2` réside dans l'utilisation d'une méthode différente pour calculer la pseudo inverse. Dans le premier cas, on utilise un solveur de moindres carrés, alors que dans le deuxième cas, on utilise la décomposition en valeurs singulières. Voici un exemple d'utilisation :

```
import numpy as np
from scipy import linalg

a = np.array([[1, 0, 0, 0, 2],
        [0, 0, 3, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 4, 0, 0, 0]])

a_h = np.array([[-1, 1 - 1j, 1 + 2j, -1j],
         [1 + 1j, 3, -2, 3 - 2j],
         [1 - 2j, -2, 0, 4],
         [1j, 3 + 2j, 4, 2]])

pinv = linalg.pinv(a)
pinv2 = linalg.pinv2(a)
pinv_h = linalg.pinvh(a_h)
```

```
print(pinv)
print(pinv2)
print(pinv_h)
```

On obtient alors :

```
[[0.2        0.         0.         0.        ]
 [0.         0.         0.         0.25      ]
 [0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.4        0.         0.         0.        ]]

[[0.2        0.         0.         0.        ]
 [0.         0.         0.         0.25      ]
 [0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.4        0.         0.         0.        ]]

[[-0.50241546+0.00000000e+00j  0.07729469-3.09178744e-01j
   0.01449275+9.66183575e-02j  0.16425121+9.66183575e-02j]
 [ 0.07729469+3.09178744e-01j -0.04830918+3.46944695e-18j
  -0.09661836+5.31400966e-02j  0.11111111-1.15942029e-01j]
 [ 0.01449275-9.66183575e-02j -0.09661836-5.31400966e-02j
   0.04830918+0.00000000e+00j  0.14975845-9.66183575e-03j]
 [ 0.16425121-9.66183575e-02j  0.11111111+1.15942029e-01j
   0.14975845+9.66183575e-03j -0.03381643-1.38777878e-17j]]
```

15 Calcul différentiel

15.1 Opérateurs différentiels

15.1.1 Diffentiel

Prenons une fonction G quelconque :

$$G = 9x^2 + 3x + 1$$

où sa différentielle est

$$g = 18x + 3$$

Passons sans tarder au code avec le module sympy. On commence par importer les modules importants, définir nos symboles/variables ainsi que notre fonction G.

```
import os
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits.mplot3d import axes3d, Axes3D
from sympy.utilities.lambdify import lambdastr


x, y = sp.symbols("x, y")
G = 9 * x ** 2 + 3 * x + 1
```

Afin de trouver la différentielle de G, on utilise la méthode "diff(expr, vars)" de sympy.

```
g = sp.diff(G, x)
```

Ensuite, on affiche G et sa dérivée sur un graphe.

```
X = np.linspace(-10, 10, 1_000)
plt.plot(X, [G.evalf(subs={"x": i}) for i in X], label=f"G = {G}")
plt.plot(X, [g.evalf(subs={"x": i}) for i in X], label=f"g := G' = {g}")
plt.grid()
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.savefig(os.getcwd() + "/diff_exemple_G_and_g.png", dpi=300)
plt.show()
```

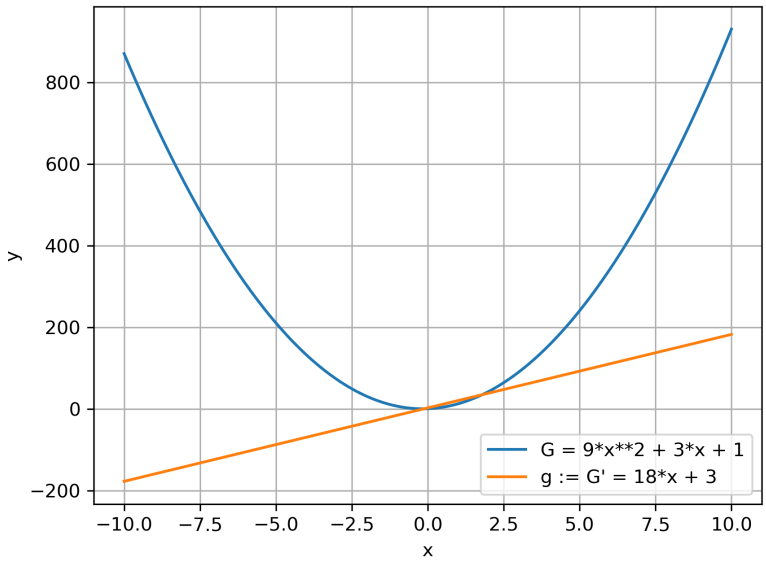


FIGURE 15.1: out

15.1.2 Gradient

Maintenant, il serait intéressant d'utiliser la méthode de différentiel afin de trouver et afficher le vecteur gradient d'une courbe. On définit une certaine fonction F,

$$F = x^2 - y^2$$

On calcule son gradient,

$$\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} \end{bmatrix}$$
$$\nabla F = \begin{bmatrix} 2x & -2y \end{bmatrix}$$

Maintenant, on met ceci en python avec sympy.

```
    F = x**2 - y**2
    grad_F = [sp.diff(F, x_i) for x_i in (x, y)]
    print(f"grad_F = {grad_F}")
out:
    grad_F = [2*x, -2*y]
```

Ensuite, on prépare nos données et on affiche notre gradient en format de fonction lambda.

```
    X = np.linspace(-5, 5, 50)
    Y = np.linspace(-5, 5, 50)
    X, Y = np.meshgrid(X, Y)
    F_lambdify = sp.lambdify((x, y), F, modules="numpy")
    print(f"Grad_F = {lambdastr((x, y), grad_F)}")
    Z = F_lambdify(X, Y)
    grad_F_lambdify = sp.lambdify((x, y), grad_F, modules="numpy")

out:
    Grad_F = lambda x,y: ([2*x, -2*y])
```

Finalement, on affiche le tout.

```
    fig = plt.figure()
    ax = Axes3D(fig)

    # Plot the surface.
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0,
                           antialiased=False)
    # add gradient
    grad_Z = grad_F_lambdify(X, Y)
    ax.quiver(X, Y, Z, grad_Z[0], grad_Z[1], F_lambdify(*grad_Z),
              length=0.2, normalize=True,
              label=f"Vecteur directionnel du gradient")

    # Customize the z axis.
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

    # Add a color bar which maps values to colors.
    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.legend()
    plt.savefig(os.getcwd() + "/diff_exemple_F.png", dpi=300)
    plt.show()
```

Voici le graphe final.

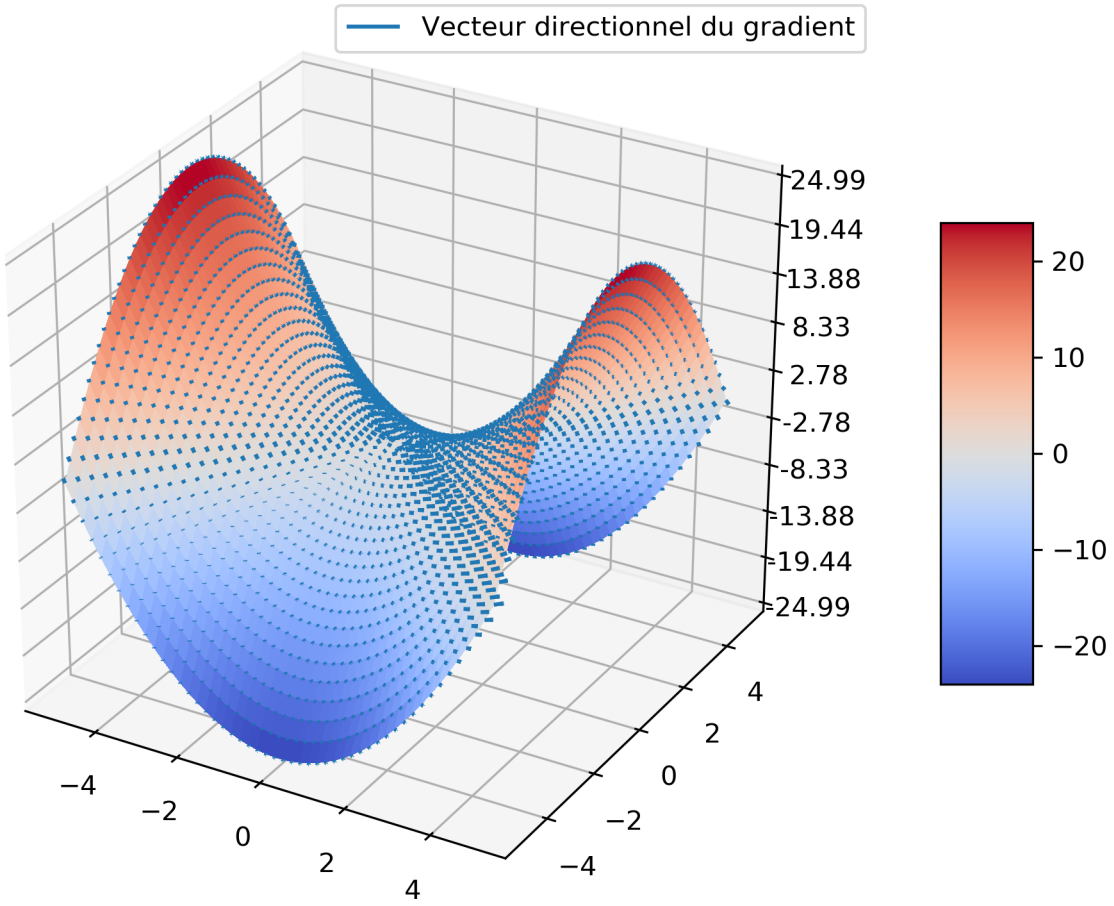


FIGURE 15.2: out

15.2 Équations différentielles

Commençons en posant une équation différentielle simple.

$$3g'(x) + x = 6$$

Résolvons,

$$\begin{aligned} 3\frac{\mathrm{d}g(x)}{\mathrm{d}x} + x &= 6 \\ \frac{\mathrm{d}g(x)}{\mathrm{d}x} &= 2 - \frac{x}{3} \\ \mathrm{d}g(x) &= 2 - \frac{x}{3}\mathrm{d}x \\ \int \mathrm{d}g(x) &= \int \left(2 - \frac{x}{3}\right)\mathrm{d}x \\ \implies g(x) &= 2x - \frac{x^2}{6} + C_1 \end{aligned}$$

Maintenant, il serait intéressant de le faire avec le calcul symbolique de sympy. Commençons par déclarer nos symboles.

```
    import sympy as sp
    from sympy.interactive.printing import init_printing


if __name__ == '__main__':
    init_printing(pretty_print=False, use_unicode=False, \
    wrap_line=False, use_latex=True)

    f, g = sp.symboles('f g', cls=sp.Function)
    x = sp.symboles("x")
```

Ensuite, on construit notre équation différentielle ordinaire avec l'objet "Eq" de sympy.

```
    edo_g = sp.Eq(3*g(x).diff(x) + x, 6)
    print(sp.pretty(edo_g))

out:
          d
    x + 3*--(g(x)) = 6
          dx
```

Finalement, on demande à sympy de résoudre notre équation différentielle à l'aide de la méthode "dsolve" et on affiche le résultat afin de le comparer à ce qu'on a obtenue plus tôt.

```
    edo_g_solved = sp.dsolve(edo_g, g(x))
    print(sp.pretty(edo_g_solved))
```

```
out:
                    2
                   x
    g(x) = C1 - ── + 2*x
                   6
```

Amusons nous avec une EDO plus complexe afin de montrer la puissance et l'utilité de sympy.

$$x^2 \frac{d^2}{dx^2} f(x) + x \frac{d}{dx} f(x) + \left(x^2 - v^2\right) f(x) = 0$$

Si nous observons bien, nous avons ici une équation de bessel d'ordre v. Regardons si sympy est capable de la résoudre.

On définie notre équation et on résout.

```
    v = sp.Symbol("v", reel=True, positive=True)
    bessel_edo = sp.Eq((x**2)*f(x).diff(x, x) + \
    x*f(x).diff(x) + (x**2 - v**2)*f(x), 0)
    print(sp.pretty(bessel_edo))

    bessel_edo_solved = sp.dsolve(bessel_edo, f(x))
    print(sp.pretty(bessel_edo_solved))

out:
     2  d              d           /   2     2\
    x *---(f(x)) + x*--(f(x)) + \- v  + x /*f(x) = 0
          2            dx
       dx
    f(x) = C1*besselj(v, x) + C2*bessely(v, x)
```

Sympy est donc capable de reconnaître que c'est une équation de bessel d'ordre v. De plus, il est capable de reconnaître une grande multitude de fonctions spéciales, on peut voir celles-ci à l'adresse suivant :

https://docs.sympy.org/latest/modules/functions/special.html

Maintenant, observons une autre équation très populaire en physique, l'oscillateur harmonique.

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}t^2} + \omega_0^2 \psi(t) = 0$$

Que nous connaissons la solution

$$\psi(t) = A\cos(\omega_0 + \phi)$$

Avec sympy,

```
    psi = sp.symboles("psi", cls=sp.Function)
    t = sp.Symbol("t", reel=True, positive=True)
    omega0 = sp.Symbol("omega_0", reel=True)
    osc_hrm_edo = sp.Eq(psi(t).diff(t, t) + (omega0**2)*psi(t), 0)
    print(sp.pretty(osc_hrm_edo))

    osc_hrm_edo_solved = sp.dsolve(osc_hrm_edo, psi(t))
    print(sp.pretty(osc_hrm_edo_solved))

out:
                        2
       2               d
    omega_0 *psi(t) + ---(psi(t)) = 0
                        2
                      dt
                 -I*omega_0*t        I*omega_0*t
    psi(t) = C1*e              + C2*e
```

où la solution

$$\psi(t) = C_1 e^{-j\omega_0 t} + C_2 e^{j\omega_0 t}$$

est équivalente à notre solution initiale.

Nous pouvons aussi nous attarder à l'oscillateur harmonique quantique.

L'EDO indépendante du temps.

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi(x) + V(x)\psi(x) = E\psi(x)$$

Où le potentiel $V(x)$ est défini comme :

$$V(x) = \frac{1}{2}m\omega^2 x^2$$

Alors nous avons l'EDO $-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi(x) + \frac{1}{2}m\omega^2x^2\psi(x) = E\psi(x)$ à résoudre.

```
psi = sp.Function("psi")
E, hbar, m, omega = sp.symboles("E hbar m omega", reel=True)
V = (1/2)*m*(omega**2)*(x**2)
quantum_osc_hrm_edo = sp.Eq((-(hbar**2)/(2*m))*psi(x).diff(x, x) \
+ V*psi(x), E*psi(x))
print(sp.pretty(quantum_osc_hrm_edo))
print(sp.latex(quantum_osc_hrm_edo))

quantum_osc_hrm_edo_solved = sp.dsolve(quantum_osc_hrm_edo, psi(x))
print(sp.pretty(quantum_osc_hrm_edo_solved))
print(sp.latex(quantum_osc_hrm_edo_solved))
```

out :

$$-\frac{\hbar^2\frac{d^2}{dx^2}\psi(x)}{2m} + 0.5m\omega^2x^2\psi(x) = E\psi(x)$$

$$\psi(x) = -\frac{Emx^5r(3)}{10\hbar^2} + C_2\left(\frac{E^2m^2x^4}{6\hbar^4} - \frac{Emx^2}{\hbar^2} + 1 + \frac{m^2\omega^2x^4}{12\hbar^2}\right) + C_1x\left(1 + \frac{m^2\omega^2x^4}{20\hbar^2}\right) + O\left(x^6\right)$$

16 Calcul intégral

16.1 Sympy

Le module sympy offre la possibilité de faire des calculs mathématiques de façon symbolique. Voici comment on peut faire des intégrales simples avec celui-ci. Disons que nous avons la fonction f suivante :

$$f = \cos(x + \theta) + \sin(x + \phi)$$

Et que nous cherchons à trouver la primitive F de la fonction f. Soit,

$$\begin{aligned} F &= \int \cos(x + \theta) + \sin(x + \phi) \\ \implies F &= \sin(\theta + x) - \cos(\phi + x) + C \end{aligned}$$

Pour ce faire, il suffit d'utiliser le code suivant :

```
    x = sp.Symbol('x')
    theta, phi = sp.symbols("theta phi")

    f = sp.cos(x + theta) + sp.sin(x + phi)
    print(f"f = {f}")

    F = sp.integrate(f, x)
    print(f"F = {F}")

out:
    f = sin(phi + x) + cos(theta + x)
    F = sin(theta + x) - cos(phi + x)
```

Nous pouvons aussi faire faire une intégrale définie, en ajoutant nos bornes d'intégration dans la fonction "integrate".

```
    F_0_to_pi = sp.integrate(f, (x, 0, np.pi))
    print(f"F_0_to_pi = {F_0_to_pi}")

out:
    F_0_to_pi = -sin(theta) + sin(theta + 3.14159265358979) + cos(phi)
    - cos(phi + 3.14159265358979)
```

Maintenant, il serait intéressant d'évaluer cette fonction pour certaines valeurs de θ et de ϕ.

```
    F_0_to_pi_evalf = F_0_to_pi.evalf(subs={theta: 0, phi: np.pi/4})
    print(f"F_0_to_pi_evalf = {F_0_to_pi_evalf}")
```

```
out:
    F_0_to_pi_evalf = 1.41421356237310
```

Nous pouvons aussi créer un objet de type "Integral" afin de générer la primitive d'une fonction et être capable d'exécuter le calcul de la primitive plus tard avec la méthode "doit()" de sympy. Ceci est pertinent quand nous voulons gérer plusieurs intégrales compliquées et donc longues à calculer.

```
    F = sp.Integral(f, x)
    print(f"F = {F}")

    F = F.doit()
    print(f"F = {F}")
```

```
out:
    F = Integral(sin(phi + x) + cos(theta + x), x)
    F = sin(theta + x) - cos(phi + x)
```

Maintenant, disons que nous avons une nouvelle fonction g plus complexe et possédant deux variables indépendantes.

$$g = \sin(x+\theta)\cos(y+\phi) + \tan(x+\theta) + \operatorname{acos}(y+\phi)$$

Trouvons G,

```
    y = sp.Symbol('y')
    g = sp.sin(x + theta)*sp.cos(y + phi) + sp.tan(x + theta)
        + sp.acos(y + phi)
    G = sp.integrate(g, x, y)
    print(f"G = {G}")
```

```
out:
    G = x*(phi*acos(phi + y) + y*acos(phi + y)
        - sqrt(-phi**2 - 2*phi*y - y**2 + 1))
        + y*log(tan(theta + x)**2 + 1)/2
        - sin(phi + y)*cos(theta + x)
```

Nous pouvons aussi faire évaluer une intégrale définie multiple en y ajoutant nos bornes d'intégration dans l'ordre que nous voulons que sympy calcule celle-ci.

```
    G_evalf = sp.integrate(g,
                (x, 0, np.pi), (y, 0, 2*np.pi)).evalf(subs={theta: np.pi/2,
                                                           phi: np.pi/6})
    print(f"G_evalf = {G_evalf}")
```

```
out:
    G_evalf = -5.90364093687815 + 34.5669841038705*I
```

Où I représente l'unité imaginaire $j = \sqrt{-1}$.

16.2 Scipy

Posons que nous avons la fonction $f(x, y, z)$ et que nous cherchons à trouver P, son intégrale définie pour certaines bornes.

$$f(x, y, z) = x^2 + y^2 + \cos(z)$$

Et nous cherchons

$$\begin{aligned} P &= \int_0^{\pi} \int_0^{5z} \int_{-10y}^{10z} x^2 + y^2 + \cos(z) \mathrm{d}x\mathrm{d}y\mathrm{d}z \\ P &= \left[c_1 xy + c_2 x + c_3 + 1/3x^3yz + 1/3xy^3z + xy\sin(z) \right] \Big|_{z=0}^{\pi} \Big|_{y=0}^{5z} \Big|_{x=-10y}^{10z} \\ \implies P &= 3.40974 \cdot 10^6 \end{aligned}$$

```
    def f(x, y, z):
        return x**2 + y**2 + sc.cos(z)

    # Ici on defini nos bornes d'integration
    def bound_x(y, z):
        return [-10*y, 10*z]

    def bound_y(z):
        return [0, 5*z]

    def bound_z():
        return [0, sc.pi]

    P = nquad(f, [bound_x, bound_y, bound_z])
    print(f"P = {P[0]}, absolute error = {P[1]}")

out:
    P = 3.40974e+06, absolute error = 5.97279e-08
```

16.3 Monte-Carlo

Maintenant, essayons avec une méthode statistique appelée Monte-Carlo. Cette méthode est simple et efficace, mais demande la plupart du temps beaucoup de ressources de calcul. Voici une brève explication de cette technique.

Prenons par exemple, la fonction $y = 3x + 2$ et nous voulons connaître son intégrale définie pour $x \in [0, 10]$. Nous pouvons en premier lieu tracer cette fonction.

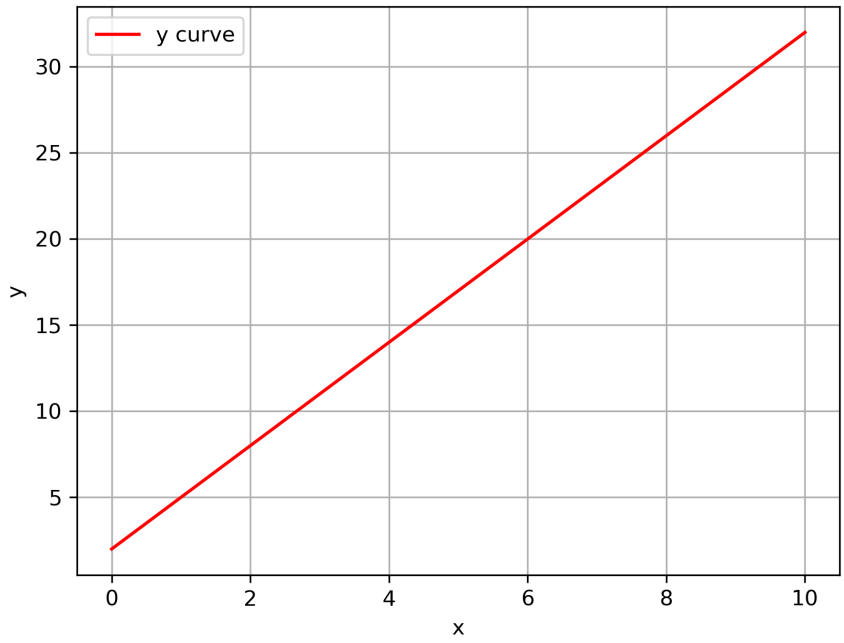


FIGURE 16.1: Fonction y

Sachant que nous voulons connaître l'aire sous la courbe illustrée à la figure 16.1, nous allons tracer un carré de base 10 en x et de hauteur $y(10) - y(0)$ (ce qui correspond à nos bornes d'intégrations). Ensuite, nous allons tirer aléatoirement un certain nombre de points, disons 100. L'objectif sera de connaître le ratio de ces points aléatoires qui sont à l'intérieur de la courbe, car ce ratio correspondra en fait au ratio que prend l'aire sous la courbe y par rapport à l'aire de notre précédent carré.

Effectuons cette expérience.

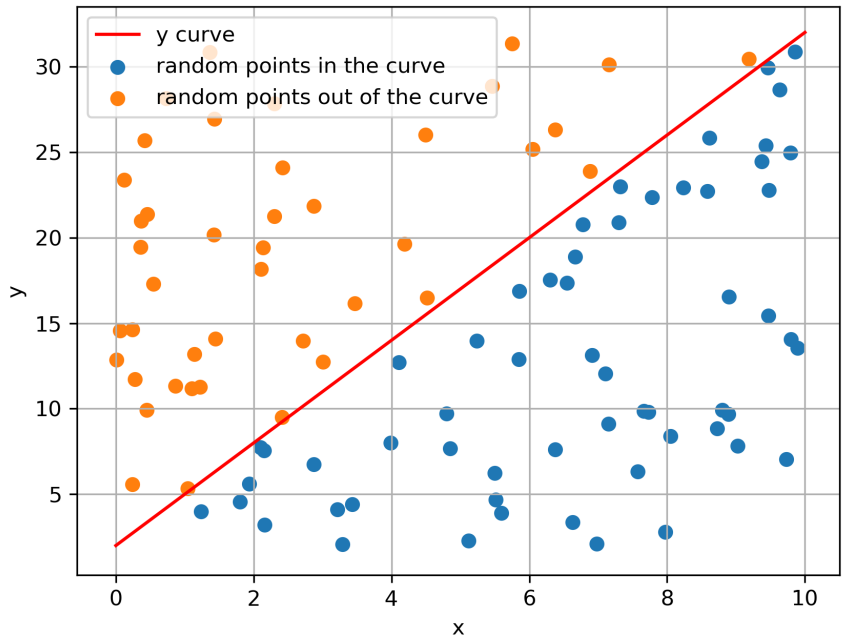


FIGURE 16.2: Fonction y avec points aléatoires

Maintenant, effectuons le calcul de P l'intégral défini de y pour $x \in [0, 10]$.

$$P = \frac{count_{in}}{count_{sample}} \cdot A$$
$$P = \frac{58}{100} \cdot 10(3(10) + 2 - 3(0) + 2)$$
$$P = 174$$

Vérifions avec un calcul de l'intégral à la main.

$$P = \int_0^{10} (3x + 2)\mathrm{d}x$$
$$P = 170$$

Ce qui est très acceptable pour un nombre aussi faible de points. Vous pourriez tester l'amélioration de la précision en augmentant le nombre de points aléatoires.

Maintenant, testons cette technique avec une fonction plus complexe et ajoutons-y un peu de code. Posons que nous avons la fonction g suivante et que nous cherchons à déterminer P l'intégrale définie pour $x \in [0, 9]$, $y \in [-10, 7]$ et $z \in [0, 10]$.

$$
\begin{aligned}
g &= 4x^3 + y^2 + \sqrt{z} \\
P &= \int_0^{10} \int_{-10}^{7} \int_0^{9} \left(4x^3 + y^2 + \sqrt{z}\right) \mathrm{d}x\mathrm{d}y\mathrm{d}z \\
P &= 1020\left(1133 + 3\sqrt{10}\right) \\
P &\approx 1.16534 \cdot 10^6
\end{aligned}
$$

Définissons une fonction capable de faire le calcul de l'intégrale définie avec la technique de Monte-Carlo.

```
import numpy as np
import sympy as sp
from sympy.utilities.lambdify import lambdastr
import itertools
import time


def monte_carlo_integration(integrand, bounds: list,
                            n_sample: int = int(1e6)) -> float:
    """
    Compute the definite integral with a monte carlo integration algorithm
    :param integrand: the function to integrate (lambda or func)
    :param bounds: list of integral bounds
    ex: [[0, 10], [0, 2], [-10, 10]] (list)
    :param n_sample: number of sample to use (int)
    :return: the integral of the integrand (float)
    """
    np.random.seed(1)
    count_in_curve: int = 0

    def sampler():
        while True:
            yield [np.random.uniform(b0, b1) for [b0, b1] in bounds]

    x_rn0, x_rn1 = next(sampler()), next(sampler())
    f_min: float = np.min([integrand(*x_rn0), integrand(*x_rn1)])
    f_max: float = np.max([integrand(*x_rn0), integrand(*x_rn1)])

    for x in itertools.islice(sampler(), n_sample):
        f: float = integrand(*x)
```

```
            # Generate random point
            f_rn: float = np.random.uniform(f_min, f_max)

            # Increase the counter
            count_in_curve += 1 if 0 <= f_rn <= f else 0

            # update the domain size
            f_min = np.min([f_min, f])
            f_max = np.max([f_max, f])

    # Compute the hyper volume v of the statistical box
    v: float = np.prod([b1 - b0 for [b0, b1] in bounds]) * (f_max - f_min)
    return (count_in_curve / n_sample) * v


if __name__ == '__main__':
    bounds_x: list = [0, 9]
    bounds_y: list = [-10, 7]
    bounds_z: list = [0, 10]

    bounds: list = [bounds_x, bounds_y, bounds_z]

    def g(x, y, z):
        return 4*x**3 + y**2 + np.sqrt(z)

    start_time = time.time()

    P: float = monte_carlo_integration(g, bounds, n_sample=int(1e6))
    print(f"P = {P:.5e}")
    print(f"--- elapse time: {time.time() - start_time} s ---")

out:
    P = 1.15682e+06
    --- elapse time: 15.752187013626099 s ---
```

Maintenant, faisons le même travail, mais en utilisant le module sympy. Nous allons définir des symboles pour les variables x, y et z pour ensuite créer notre ancienne fonction g que nous allons appeler f ici. Finalement, nous allons transformer notre expression symbolique f en expression lambda afin de pouvoir la passer à notre fonction "monte_carlo_integration". De plus, nous allons tester notre méthode d'intégration pour divers nombre de points afin de comparer la précision et le temps de calcul.

```
    x, y, z = sp.symbols("x, y, z")
    f = 4*x**3 + y**2 + sp.sqrt(z)
    print(f"f = {f}")
    f_lambdify = sp.lambdify((x, y, z), f)
    print(f"f_lambdify -> {lambdastr((x, y, z), f)}")
```

```
    for n in [int(1e1), int(1e3), int(1e6), int(1e8)]:
        start_time = time.time()
        P: float = monte_carlo_integration(f_lambdify, bounds, n_sample=n)
        print(f"P = {P:.5e} for {n:2e} samples")
        print(f"--- elapse time: {time.time() - start_time} s ---")

out:
    f = 4*x**3 + y**2 + sqrt(z)
    f_lambdify -> lambda x,y,z: (4*x**3 + y**2 + sqrt(z))
    P = 3.01664e+05 for 1.000000e+01 samples
    --- elapse time: 0.0 s ---
    P = 1.24565e+06 for 1.000000e+03 samples
    --- elapse time: 0.01999950408935547 s ---
    P = 1.15682e+06 for 1.000000e+06 samples
    --- elapse time: 24.719645500183105 s ---
    P = 1.15871e+06 for 1.000000e+08 samples
    --- elapse time: 3062.00985956192 s ---
```

17 Deep learning - théorie

17.1 Idée

Matière devenant de plus en plus populaire depuis le début de ce siècle, l'apprentissage profond est une méthode permettant de résoudre des problèmes extrêmement abstraits comme la compréhension du langage humain, compréhension de signaux sonores, compréhension des mathématiques, la reconnaissance faciale et vocale, la reconnaissance d'images, la classification d'images et bien d'autres.

Les algorithmes d'apprentissages profonds ont pour objectif final de passer de valeur en "input" à des ou une valeur(s) en "output". Le processus intermédiaire afin de passer l'un à l'autre est souvent comparé comme étant une boîte noire puisqu'il est extrêmement difficile de savoir exactement se qui se passe à l'intérieur. Dans cette section, nous allons tout de même tenter d'éclaircir le plus possible ce qui se passe dans cette fameuse boîte noire.

$$f(input) = output$$

L'apprentissage profond se base sur l'optimisation de poids d'une fonction non linéaire. Commençons avec un exemple simple avec une fonction linéaire afin de bien comprendre la logique. Prenons par exemple la fonction suivante :

$$y = \alpha x + \beta$$

Posons $\beta = 10$ afin de simplifier encore plus les prochaines étapes. De plus, nous allons poser initialement que $\alpha = 0$ puisque nous n'avons aucune idée de l'ordre de grandeur de sa vraie valeur. Considérons l'ensemble de données suivant :

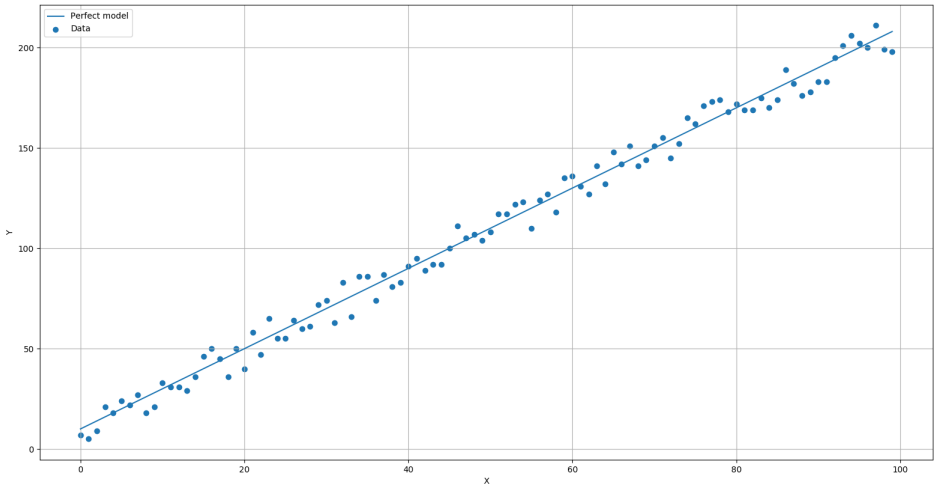


FIGURE 17.1: Ensemble de données du mini exemple

Voici l'objectif : se fabriquer un modèle qui optimisera le paramètre α afin de se rapprocher le plus possible du modèle parfait en ayant seulement les données à notre disposition.

Notre stratégie pour arriver à notre objectif sera la suivante. Nous allons créer un modèle qui comme mentionné plus tôt prédira les données de cette façon : $y = 0 \cdot x + 10$.

```python
class Model:
    def __init__(self):
        self.alpha = 0
        self.beta = 10

    def optimize(self, loss, lr):
        if loss > 0:
            self.alpha += lr
        elif loss < 0:
            self.alpha -= lr

    def __call__(self, x):
        return self.alpha * x + self.beta


def train(epochs, data, model, plot_loss):
    lr = 0.001
    losses = []
    for _ in range(epochs):
        running_loss = 0
        for i, (input, target) in enumerate(data):
            output = model(input)
            loss = target - output
            running_loss += loss
            model.optimize(loss, lr)
        losses.append(running_loss/len(data))
    if plot_loss:
        plt.plot(range(epochs), losses)
        plt.title("training loss")
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.grid()
        plt.show()
```

Ensuite, nous itérerons sur toutes les données en calculant l'écart moyen entre les données et la courbe prédite par notre modèle. Une fois fait, nous allons optimiser le paramètre α de notre modèle avec l'écart calculé afin de réduire celui-ci la prochaine fois que nous allons itérer sur toutes le données. Ensuite, nous recommençons ce processus jusqu'à tant que l'écart soit assez minime pour qu'elle cesse de changer entre-deux «epochs». Un «epoch» est le processus que nous venons de mentionner, c'est le fait d'itérer sur toutes nos données d'entraînement en optimisant les paramètres de notre modèle. Le modèle est comme vous avez pu comprendre, l'objet qui apprend à prédire le mieux possible les données d'entraînement.

Si nous procédons à l'entraînement de notre modèle de la manière décrite précédemment, voici les résultats obtenus :

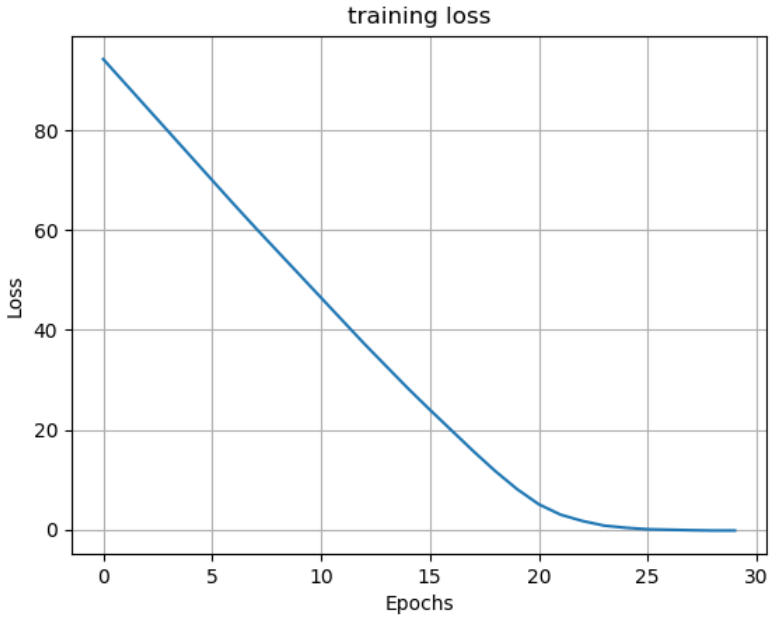


FIGURE 17.2: Training Loss mini exemple

Ici, nous pouvons voir que la perte d'entraînement devient nulle à environ 25 «epochs» ce qui veut

dire qu'à ce moment notre modèle à réussi à être le modèle parfait pour prédire ces données.

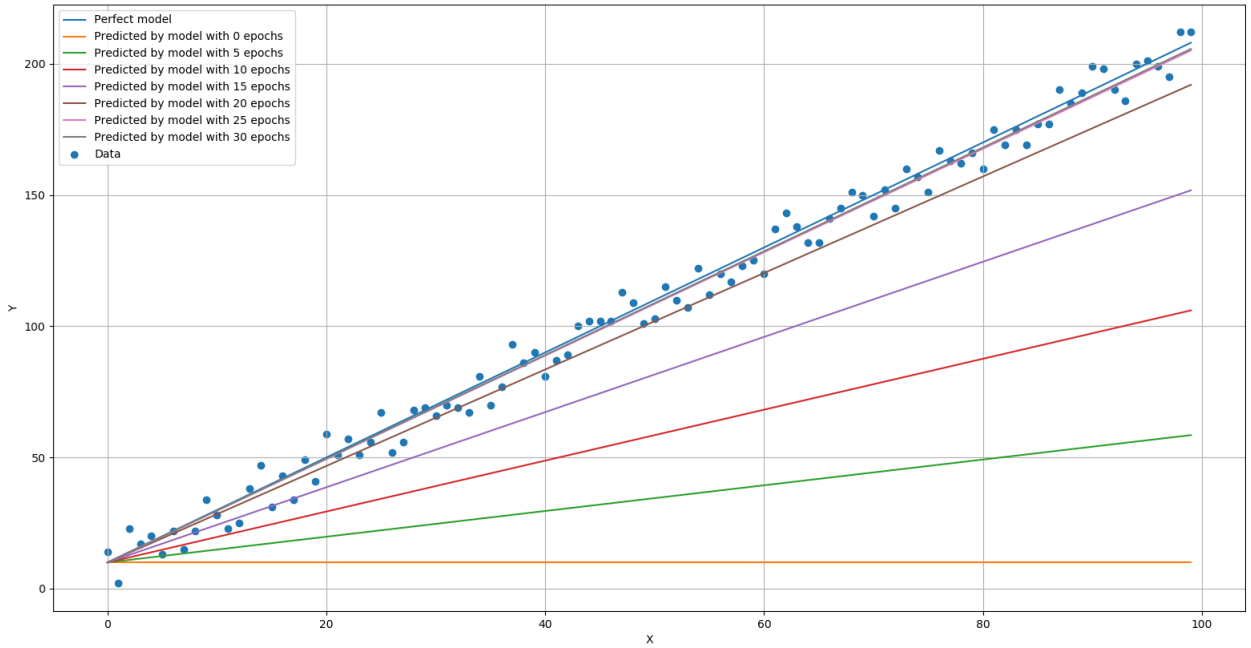


FIGURE 17.3: Prédictions du modèle pour le mini exemple

Ici, nous pouvons voir notre modèle évoluer avec le temps pour finalement prédire de la meilleure façon possible les données. Vous venez donc de comprendre sur quoi se base l'apprentissage profond.

Maintenant que nous avons vu un petit exemple avec un seul paramètre à optimiser, entrons dans le vif du sujet avec un autre exemple beaucoup plus concret. Le classificateur MNIST. Prenons les données suivantes comme données d'entraînement :

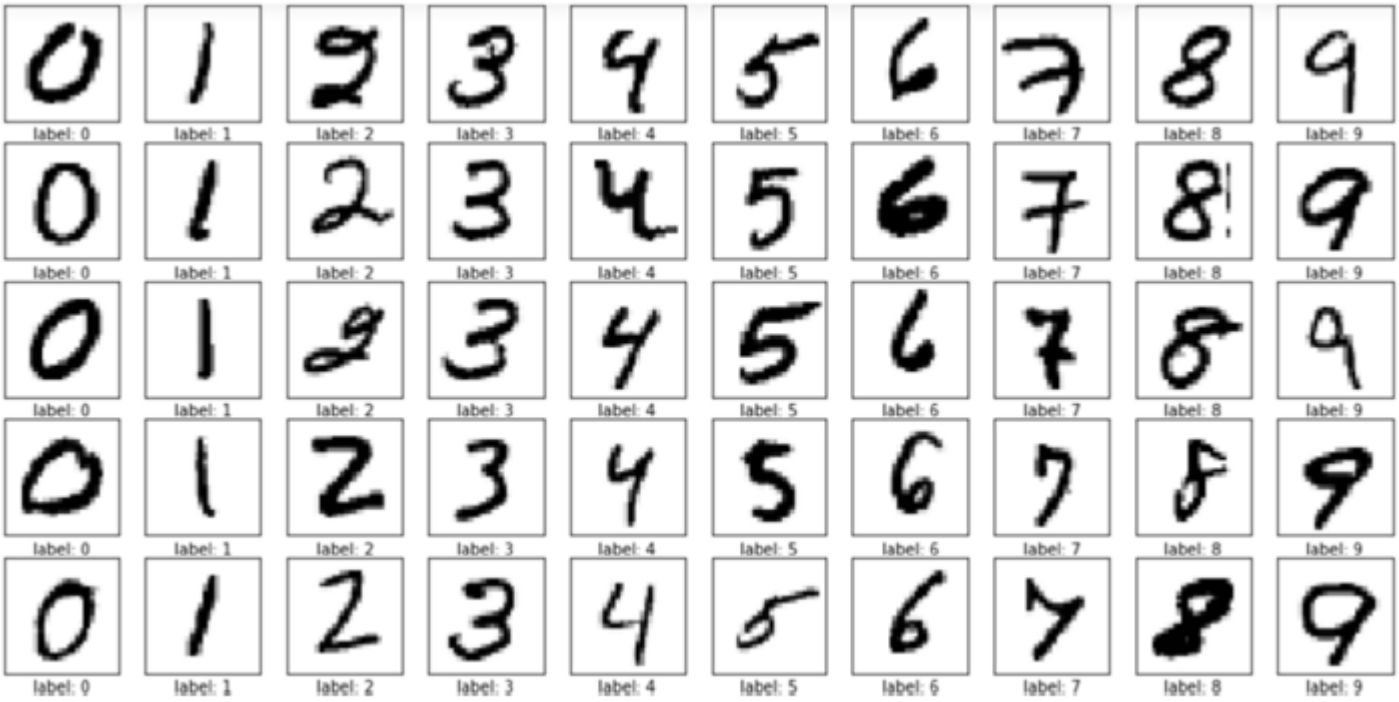


FIGURE 17.4: Exemple de données d'entraînement de MNIST

Nous pouvons voir que nous devons classifié 10 types d'images avec chacune d'entre elles ayant, disons n pixels. Commençons par trouver la forme de notre fonction de prédiction. Disons que nous voulons un paramètre par pixel et par classe. Donc notre fonction pourrait être la suivante :

$$f(image; W, b) = V$$
$$f = W \cdot x + b$$

Dans ce cas, W est une matrice n par c le nombre le classe (10). Les paramètres représentés par b sont un vecteur 1 par c. Le vecteur d'«input» x est un vecteur de pixels représentant l'image 1 par n.

Prenons par exemple le cas où l'image possède seulement 4 pixels et que la matrice W est initialisée aléatoirement. Dans cet exemple, l'image sera par exemple un 2.

$$W \cdot x + b = output$$
$$\begin{pmatrix} 0.7 & -0.3 & 0.2 & 0.3 \\ -0.5 & 1.7 & 1.5 & 0.4 \\ 0.7 & 1.1 & -0.4 & -0.1 \\ -0.5 & 0.2 & 1.6 & 0.1 \end{pmatrix} \cdot \begin{pmatrix} 0.1 \\ 0.4 \\ 0.3 \\ 0.7 \end{pmatrix} + \begin{pmatrix} 1.0 \\ 1.2 \\ -0.4 \\ -2.1 \end{pmatrix} = \begin{pmatrix} 1.22 \\ 2.56 \\ -0.8 \\ -1.52 \end{pmatrix}$$

On peut voir que l'«output» possède un argument plus grand que les autres. Celui-ci s'agît de la prédiction de ce modèle. En effet, l'«output» est un vecteur de probabilité, donc la probabilité la plus élevée est la prédiction. Dans ce cas, la probabilité la plus élevée est à l'index (commençant à 1) 2. C'est donc dire que le modèle prédit que l'image à l'entrée est un 2. Ce qui est tout à fait vrai. Les matrices W et b semblent posséder de bons paramètres pour trouver le chiffre 2.

Il est temps d'implémenter cette méthode avec pytorch. Considérons des images 28x28 donc 784 pixels d'entrées, une matrice W de 10x784 et un b de 10x1. En apprentissage profond ceci correspond à une couche linéaire de 784 en entrée et 10 en sortie. Avec un lot de 60 000 exemples d'entraînement et 10 000 exemples de validation, on peut avoir jusqu'à environ 92% de précision. En apprentissage profond, il est bien connu que plus un modèle est profond (plus il possède de couches) plus il sera possible pour le modèle de posséder du pouvoir d'expression afin d'effectuer une tâche complexe. Nous pourrions alors penser qu'ajouter une couche à notre présent modèle pourrait augmenter la précision du modèle. Dans les faits, en ajoutant seulement une autre couche, le modèle n'aura pas plus de pouvoir d'expression pour la raison suivante :

```
class SimpleLinear(nn.Module):
    def __init__(self):
        super(SimpleLinear, self).__init__()
        self.fc1 = nn.Linear(784, 10)

    def forward(self, x):
        x = x.view(-1, 784) # Flatten
        x = self.fc1(x)
        return x
```

$$
\begin{aligned}
W_1 \cdot x + b_1 + W_2 \cdot x + b_2 &= output \\
(W_1 + W_2) \cdot x + b_1 + b_2 &= output \\
\text{posons } W_1 + W_2 = W_3 \text{ et } b_1 + b_2 &= b_3 \\
W_3 \cdot x + b_3 &= output
\end{aligned}
$$

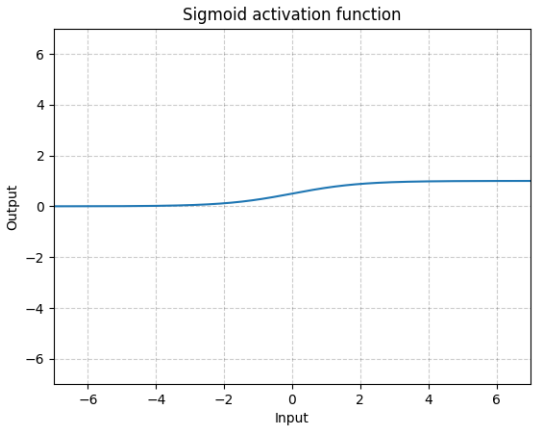
Les deux couches n'augmentent pas le pouvoir d'expression puisqu'elles se simplifient. Afin de pallier à ce problème, il faut ajouter des non-linéarités entre les couches, ce qui empêchera la simplification et augmentera le pouvoir d'expression du modèle.

17.2 Fonctions d'activations

Les fonctions d'activations apportent de la non-linéarité aux réseaux de neurones en apprentissage profond. En voici quelques-unes de base.

17.2.1 Sigmoïde

La sigmoïde est une des premières fonctions qui n'est plus beaucoup utilisée. Dorénavant, elle est surtout utilisée en sortie réseau afin de permettre que les valeurs en sortie soient entre 0 et 1. Ce qui reflète plus une probabilité. Surtout utilisée aussi en sortie de réseau pour la classification multiple. C'est lorsqu'on essaie de prédire l'appartenance à non seulement une classe, mais plusieurs en même temps. Autrement, elle n'est pas vraiment utilisée. Elle utilise l'équation suivante :



$$\sigma(z) = \frac{1}{1 + e^{-x}}$$

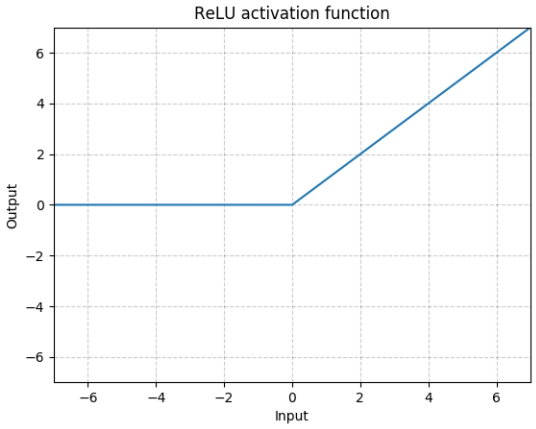
17.2.2 Softmax

La softmax est une fonction tout de même très utilisée et surtout en sortie de réseau pour la prédiction de classe. Par exemple, nous aurions pu utiliser la softmax en sortie du petit modèle créé dans l'exemple de MNIST puisque nous tentions de prédire l'appartenance d'une image à une seule classe. En fait, elle ramène les valeurs en sortie entre 0 et 1 et où la somme de ces valeurs est 1. Elle tente alors de représenter la probabilité d'une entrée d'appartenir à chacune des classes. Voici son expression :

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

17.2.3 ReLu

La ReLu est la fonction d'activation la plus populaire dans l'apprentissage profond. Elle est surtout utilisée dans les couches intermédiaires des réseaux neurones. Nous aurions pu par exemple, l'utiliser entre nos deux couches linéaires de l'exemple de MNIST afin d'ajouter la non-linéarité qu'il manquait. Cette fonction d'activation a révolutionné l'apprentissage profond, non seulement puisqu'elle permet une descente de gradient approprié, mais par sa simplicité légendaire. Effectivement, elle est vraiment très simple, ce qui permet même la complexification des modèles d'apprentissage profond. Voici son expression :



$$ReLu(x) = max(0, x)$$

17.3 Types d'apprentissages

En apprentissage machine, il est important d'avoir un bon modèle puisque c'est celui-ci qui possède le pouvoir de réussir le travail qu'on lui demande de faire. Toutefois, un modèle sans entraînement possède la même performance qu'un algorithme procédant à des actions complètement aléatoires. Un modèle mal entraîné peut sembler excessivement bon, mais une fois qu'on le teste, il peut devenir aussi ou même moins performant que l'aléatoire. Dans ce cas il est facile de comprendre l'importance de l'entraînement et que l'essentiel de la difficulté en apprentissage profond réside dans cette partie.

On peut catégoriser les algorithmes d'apprentissage par leur type d'entraînement, voici les trois principaux.

17.3.1 Supervisé

L'apprentissage supervisé est le type le plus intuitif. Considérant que nous avons un conteneur de données appelé C. Ce type d'apprentissage consiste à fournir au modèle une certaine donnée et lui demander de prédire une certaine étiquette préalablement choisie par un expert. Le conteneur C renvoie donc le tuple $(donnee, etiquette)$ lors de son appel durant l'entraînement. Le tout est fait avec l'objectif de prédire ces étiquettes lors de l'application du modèle afin de remplacer l'expert. Un bon exemple de ce type d'apprentissage est l'exemple donné précédemment avec le modèle de MNIST. En effet, on lui donnait le tuple $(image, chiffre)$ en entraînement dans le but que le modèle soit capable de faire les bonnes associations afin de remplacer un lecteur.

17.3.2 Non supervisé

Il s'agit du type d'apprentissage opposé à l'apprentissage supervisé. Ici, au lieu de donner la réponse au modèle, on lui demande de faire la corrélation entre les données d'entraînements sans support d'un expert. Le meilleur exemple de ce type d'apprentissage doit être le partitionnement de données ou plus communément appelé ¡¡clustering¿¿. En effet, le partitionnement de données se fait de façon non supervisée, on demande au modèle de trouver lui-même la structure cachée des données afin de regrouper les données dans des sous-groupes possédant les mêmes caractéristiques.

17.3.3 Par renforcement

Ce type d'apprentissage est à mon avis le plus intéressant. Ici, on ne parle plus de modèle, mais d'un agent. L'agent ne doit pas s'entraîner sur une banque de données, mais doit apprendre à réagir de façon efficace à son environnement. Durant son entraînement, il reçoit séquentiellement l'état de son environnement et il effectue une action. Une fois l'action effectuée, il reçoit une récompense reliée au résultat que son action a fait à l'état de son environnement. Tout au long de l'entraînement, l'agent met à jour sa mémoire, son conteneur de données, qui est constitué des états successifs de l'environnement reliés à l'action prise par l'agent dans le passé et la récompense associée. Il doit repasser sur toute sa mémoire afin de faire le lien entre ses actions, son environnement et les récompenses. L'agent s'améliore alors au fil du temps dans le but d'obtenir les meilleures récompenses possible. Ce type d'entraînement est surtout utilisé dans les jeux vidéos afin qu'un agent apprenne à réussir les niveaux les plus difficiles que peut posséder un jeu.

17.4 Fonctions de pertes

La fonction de perte en apprentissage machine est excessivement importante, puisqu'elle informe le modèle sur sa performance durant l'entraînement. Le but ultime de l'entraînement d'un modèle est de faire tendre le plus possible sa perte ou on pourrait tout aussi dire son imprécision vers 0. Par contre, si la perte n'est pas calculée de la bonne façon elle peut faire diverger un modèle et le rendre inapte à prendre de bonnes décisions. Il y a essentiellement deux types de prédictions qu'un modèle peut faire : prédire une ou des classes (classification) ou bien prédire une ou des valeurs (régression).

17.4.1 Classification

Lorsqu'on tente de prédire une classe unique, nous devons avoir une fonction de perte adaptée en conséquence. Celle la plus utilisée pour ce genre de tâche est sans aucun doute la ¡¡CrossEntropyLoss¿¿ et son calcul [28]de perte peut se décrire comme suit :

$$loss(x, class) = -log\left(\frac{e^{x[class]}}{\sum_j e^{x[j]}}\right)$$

$$\implies loss(x, class) = -x[class] + log\left(\sum_j e^{x[j]}\right)$$

Il existe aussi la prédiction multiclasse qui consiste à prédire les classes auxquelles la donnée est associée. Dans ce cas, nous ne pouvons pas utiliser la ¡¡CrossEntropyLoss¿¿. On peut donc utiliser la ¡¡BCELoss¿¿ qui est très populaire pour faire ce type de travail. Son calcul [29]de perte est donné par l'expression suivante.

$$\ell(x, y) = L = \{\ell_1, ...\ell_N\}^T$$
$$\text{where } \ell_n = -w_n[y_n \cdot log(x_n) + (1 - y_n) \cdot log(1 - x_n)]$$

Où N est la taille de la ¡¡batch¿¿ et où $\ell(x, y)$ peut être $mean(L)$ ou $sum(L)$ selon ce qui est voulu.

17.4.2 Régression

Lorsqu'on tente de faire une régression vers des valeurs données, on va majoritairement utiliser la fonction de perte ¡¡MSELoss¿¿ qui est la plus populaire de toutes les fonctions de perte pour la régression. La ¡¡MSELoss¿¿ a aussi la particularité de pouvoir s'appliquer à tous les problèmes. En effet, dû à son implémentation très simple, elle peut faire autant de la classification que de la régression, mais n'est pas optimisée pour cette première. Voici son calcul [30] :

$$\ell(x, y) = L = \{\ell_1, ...\ell_N\}^T$$
$$\text{where } \ell_n = (x_n - y_n)^2$$

Où N est la taille de la ¡¡batch¿¿ et où $\ell(x, y)$ peut être $mean(L)$ ou $sum(L)$ selon ce qui est voulu.

Sinon il existe la ¡¡L1Loss¿¿ [31]qui est sensiblement pareil que la précédente et est légèrement moins utilisé :

$$\ell(x, y) = L = \{\ell_1, ...\ell_N\}^T$$
$$\text{where } \ell_n = |x_n - y_n|$$

Où N est la taille de la ¡¡batch¿¿ et où $\ell(x, y)$ peut être $mean(L)$ ou $sum(L)$ selon ce qui est voulu.

17.5 Optimiseurs

Il est certain qu'il est excessivement important de calculer la perte des prédictions durant l'entraînement, mais une fois fait, il faut faire quelque chose avec cette valeur. En effet, à chaque fois

28. PyTorch CrossEntropyLoss, `https://pytorch.org/docs/stable/nn.html#crossentropyloss`
29. PyTorch BCELoss, `https://pytorch.org/docs/stable/nn.html#bceloss`
30. PyTorch MSELoss, `https://pytorch.org/docs/stable/nn.html#mseloss`
31. PyTorch L1Loss, `https://pytorch.org/docs/stable/nn.html#l1loss`

que nous obtenons une perte associée à une prédiction, il faut optimiser les poids contenus dans le modèle afin que celui-ci améliore ses prédictions. Il existe une multitude d'algorithmes [32]pour faire cette optimisation, mais d'entre eux se démarque particulièrement : ¡¡SGD¿¿ et ¡¡Adam¿¿. Le ¡¡SGD¿¿ (¡¡stochastic gradient descent¿¿) est connu pour être très performant et surtout stable contrairement à ¡¡Adam¿¿ qui est considéré par beaucoup le plus performant, mais considéré comme instable.

17.6 Types d'architectures

En apprentissage profond, il existe une grande variété d'architectures possibles, mais voici les trois plus populaires souvent réutilisés dans les autres types de modèles.

17.6.1 Architecture linéaire

L'architecture linéaire est la plus utilisée en apprentissage profond, puisqu'elle est presque toujours implémentée avec les autres types de modèles en sortie de réseaux. Elle est excessivement simple à utiliser et permet d'effectuer une opération non linéaire sur son entrée. Cette architecture est constituée de couches de perceptrons. Un perceptron est un opérateur agissant sur toutes ses entrées afin de faire une certaine prédiction. L'opération effectuée est une simple somme pondérée. Les poids que l'on dit toujours devoir optimiser en apprentissage profond se trouvent là, dans cette pondération de chaque perceptron. Les perceptrons forment les couches du réseau de neurones. Il existe essentiellement trois types de couches ; la couche d'entrée de réseau, la couche de sortie de réseau et les couches cachées. Ces dernières servent à augmenter le pouvoir d'expression du réseau ou autrement dit servent à augmenter la puissance de prédiction du réseau.

Voici ce à quoi un ¡¡Multi Layer Perceptron¿¿ ressemble :

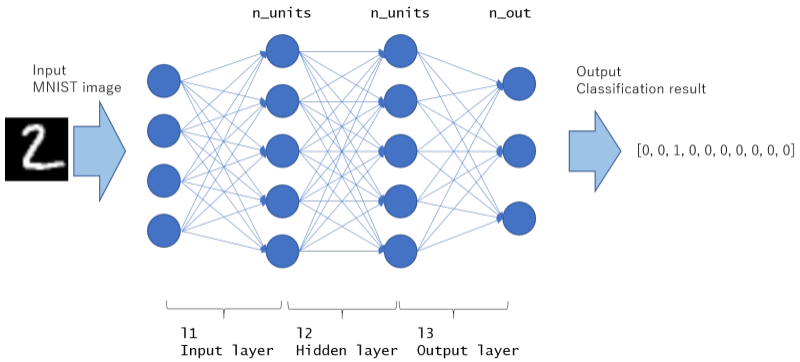


FIGURE 17.5: Exemple d'architecture linéaire avec MNIST

17.6.2 Architecture de convolution

Bien qu'efficaces pour le traitement d'images, les perceptrons multicouches (MLP) ont des difficultés à gérer des images de grande taille, en raison de la croissance exponentielle du nombre de

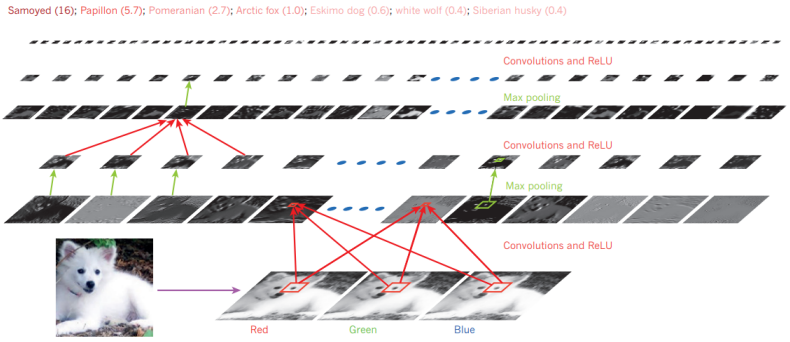
32. PyTorch optimizer, `https://pytorch.org/docs/stable/optim.html#algorithms`

connexions avec la taille de l'image, du fait que chaque neurone est « totalement connecté » à chacun des neurones de la couche précédente et suivante. Les réseaux de neurones convolutifs, dont le principe est inspiré de celui du cortex visuel des vertébrés, limite au contraire le nombre de connexions entre un neurone et les neurones des couches adjacentes, ce qui diminue drastiquement le nombre de paramètres à apprendre. Pour un réseau profond tel que AlexNet par exemple, plus de 90% des paramètres à apprendre sont dus aux 3 couches « complètement connectées » les plus profondes, et le reste concerne les (5) couches convolutives.

Dans le cadre de la reconnaissance d'image, cette dernière est « pavée », c'est-à-dire découpée en petites zones (appelées tuiles). Chaque tuile sera traitée individuellement par un neurone artificiel (qui effectue une opération de filtrage classique en associant un poids à chaque pixel de la tuile). Tous les neurones ont les mêmes paramètres de réglage. Le fait d'avoir le même traitement (mêmes paramètres), légèrement décalé pour chaque champ récepteur, s'appelle une convolution. Cette strate de neurones avec les mêmes paramètres est appelée « noyau de convolution ».

Les pixels d'une tuile sont analysés globalement. Dans le cas d'une image en couleur, un pixel contient 3 entrées (rouge, vert et bleu), qui seront traitées globalement par chaque neurone. Donc l'image peut être considérée comme un volume, et notée par exemple 30 x 10 x 3 pour 30 pixels de largeur, 10 de hauteur et 3 de profondeur correspondant aux 3 canaux rouge, vert et bleu. De manière générale, on parlera de « volume d'entrée ». [33]

Voici un aperçu d'une architecture convolutive :



Tiré de *Y. LeCun, Y. Bengio et G. Hinton, Deep Learning, Nature, vol. 521, 28 mai 2015.*

FIGURE 17.6: Exemple d'architecture de convolution

17.6.3 Architecture récurrente

Les réseaux linéaires et convolutifs sont très bons pour effectuer des tâches de traitement sur des banques de données et sur des images. Toutefois, ils ne sont pas optimisés pour un autre type de données très populaires, les séries temporelles. Les réseaux récurrents sont surtout utilisés pour la reconnaissance automatique de la parole ou de l'écriture, en traduction, le traitement des séries

33. Bonne introduction à la théorie d'un réseau convolutif, `https://fr.wikipedia.org/wiki/R%C3%A9seau_neuronal_convolutif`

bancaires, ou bien l'inférence de la connectivité de réseaux de neurones. Ce type de réseaux est très complexe, mais voici une brève description.

Un réseau récurant est un réseau de multitudes couches complétées d'un ensemble d'unités de contexte. Les couches cachées sont connectées à ces unités de contexte, et à certains poids. À chaque étape, l'entrée est retransmise et une règle d'apprentissage est appliquée. Les connexions finales fixées enregistrent une copie des valeurs précédentes des unités cachées dans les unités de contexte, puisqu'elles se propagent avant que la règle d'apprentissage soit appliquée. Donc le réseau peut maintenir une sorte d'état, lui permettant d'exécuter des tâches telles que la prédiction séquentielle qui est au-delà de la puissance d'un perceptron multicouche standard.

Voici un schéma qui illustre une architecture récurrente typique :

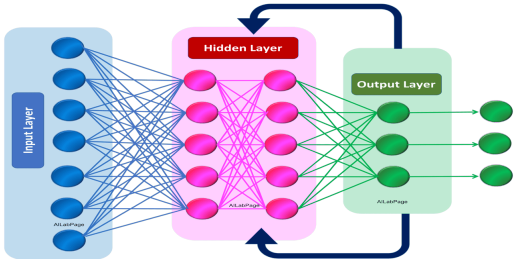


FIGURE 17.7: Exemple d'architecture récurrente

18 Solutions aux exercices

18.1 Objets built-ins

18.1.1 Types de bases

Résultat des opérations et type :

— `a + b` *#= 24.0* et le type est **`float`**

— `a + b` *#= 100.1* et le type est **`float`**

— `b % a` *#leve une exception, modulo pas defini pour complex*

Valeur manquante et type :

— `a = −12.3 + 6` et le type est **`complex`** (on peut aussi passer par `a = complex(−12.3, 6)`)

— `a = 3.0` et le type est **`float`**. Même si la division est «entière», **`float`** `//` **`int`** ou **`int`** `//` **`float`** donne un **`float`** et laisse le .0 après l'entier.

— Dans ce cas, `a` peut prendre une infinité de valeurs. En effet, $a\%2 = 0$ pour toute valeur de a égale à $2n$ où n est un entier.

18.1.2 Les strings

Affichage :

— `Hello World`

— `bonjour`

— `4`

—
```
1
2
3
4
5
6
```

La fonction `split("")` sépare la chaîne initiale en sous-chaînes selon un séparateur, ici `" "` (espace).

— `False`, mais il pourrait être tentant de répondre l'inverse, car la méthode `strip()` agit presque comme la méthode `replace("", "")`, où l'on remplace les espaces dans la chaîne par rien, on les enlève. Par contre, `strip()` le fait seulement sur les extrémités de la chaîne, soit au début et à la fin (et retire aussi d'autres caractères, comme les tabulations, changement de lignes, etc.), alors que, dans le cas présent, `replace("", "")` remplace partout, donc l'espace entre les deux mots, d'où le fait que les deux chaînes ne sont plus égales.

— Cette expression, `a[0] = "w"`, lève une exception, car les **`str`** ne peuvent être modifiés. Dans tous les cas précédents où l'on «modifie» une chaîne, on en crée une nouvelle à chaque fois, on ne modifie pas l'originale.

— `False`

18.1.3 Les listes

Éléments de la liste et type :

— Le premier élément de la liste est `[a + 12, a]`, soit une liste. Ses éléments sont `135` et `123`, deux **`int`**. Le second élément est `b + c`, ce qui équivaut à `"totoest malade"`, soit un **`str`**. Finalement, le dernier élément est **`str`**`(d)`, ce qui est à première vue **`str`**`(4j + 33)`, donc `"(4j + 33)"`.

Explications d'expressions.

— On insère les éléments de la liste `[2, 3]` dans la liste originale, tout en remplaçant le premier élément du *slicing*. Ici, l'élément à la deuxième position de la liste sera remplacé par le nouveau `2`, puis on insère le `3` par la suite. La nouvelle liste est `[1, 2, 3, 3, 4, 5]`

— On supprime le premier élément de la liste, de sorte qu'on a `[1, 2, 3, 4, 5]`

— On assigne `liste` à `liste2` de sorte que les deux variables réfèrent à la même liste. Sachant cela, lorsqu'on modifie `liste` avec `liste[0] = −1`, on modifie aussi `liste2`. Les listes intermédiaires sont `[−1, 2, 3, 4]` *#liste = liste2*. Ensuite, lorsqu'on fait `liste2[0] = 1`, on se retrouve avec la valeur initiale des listes, soit `[1, 2, 3, 4]` *#liste = liste2*

— En écrivant `liste2 = liste.copy()`, on effectue une copie de surface de `liste`. Ainsi, `liste2` contient une liste identique à `liste`, mais les deux variables ne réfèrent pas au même objet (jusqu'à une certaine limite). Ainsi, après avoir fait `liste[−1] = −1000`, on ne modifie que `liste`, alors que lorsqu'on fait `liste2[−1] = 1e34`, on ne modifie que `liste2`. Au final, on a `[1, 2, 3, −1000]` *#liste* et `[1, 2, 3, 1e+34]` *#liste2*.

— Encore une fois, on effectue une copie de surface avec la ligne `liste2 = liste.copy()`, donc lorsqu'on fait `liste[0] = −1`, on ne modifie que `liste`. Les listes sont alors `[−1, [1, 2, 3, 4], 3, 4]` *#liste* et `[1, [1, 2, 3, 4], 3, 4]` *#liste2*. Par contre, comme il s'agit d'une copie de surface, les objets composés (comme une liste (!)) dans la liste originale peuvent être modifiés par `liste` et par `liste2`. Ainsi, lorsqu'on écrit `liste[1][:] = "t"`, d'une part, on accède au second élément de `liste`, soit une liste. Puis, en faisant `[:]`, on «prend» tous les éléments de la liste. Finalement, `= "t"` remplace tous les éléments par `"t"`. On ne remplace pas chaque élément pas la tabulation, mais on remplace tout par la tabulation. Ainsi, comme il s'agit d'une modification d'un objet composé dans un objet composé copié en surface, les deux listes se trouvent modifiées. Au final, on a `[−1, ["t"], 3, 4]` *#liste* et `[1, ["t"], 3, 4]` *#liste2*.

— La première ligne est une instanciation de liste par compréhension. On utilise une formule mathématique pour créer la liste. Ensuite, comme il a été mentionné au point précédent, utiliser la **`slice`** `[:]` permet d'accéder à tous les éléments d'une liste (on de tout autre objet supportant **`slice`**). En faisant **`del`** `liste[:]`, on se retrouve à supprimer tous les éléments de la liste. Finalement, la liste est `[]`.

— On fait encore une copie de surface. Ensuite, on supprime entièrement la variable `liste`. L'affichage de `liste2` se fait sans problème. Rien n'est modifié.

— Il s'agit ici d'assignation bête, donc les deux listes réfèrent au même objet. En supprimant `liste`, on pourrait croire que `liste2` se retrouve affecté, mais ce n'est pas le cas. Le contenu de `liste2` reste intact. *Python* gère bien sa mémoire dans ce cas. Un objet en mémoire n'est supprimé que lorsqu'il n'y a aucune variable l'utilisant (ou lui faisant référence). Ainsi, lorsqu'on fait **`del`** `liste`, on ne fait que «détruire» le lien qui unit la variable à son contenu. Donc, l'objet n'est pas détruit. On peut y accéder normalement.

— Pas mal la même explication qu'au point précédent.

18.1.4 Les autres objets built-ins

Construire un dictionnaire à partir d'une liste de clés et de valeurs :

— On peut utiliser `dictionnaire = dict(zip(clefs, valeurs))`. La fonction **zip** prend des itérateurs (ici, deux listes), et retourne une sorte de générateur dont chaque élément est un tuple. Par exemple, si on a n itérables et chaque itérable a i éléments, on aura au final i tuples. Le premier tuple contient le premier élément de chaque itérable (dans l'ordre d'insertion dans les paramètres de la fonction), le deuxième contient le deuxième élément de chaque itérable et ainsi de suite.

On pourrait aussi procéder avec :

```
dictionnaire = {}

for i in range(len(clefs)):
    dictionnaire[clefs[i]] = valeurs[i]
```

Ou on pourrait faire :

```
dictionnaire = dict([(clefs[i], valeurs[i]) for i in range(len(clefs))])
```

Il y aussi d'autres solutions possibles, comme assigner manuellement les éléments aux bonnes clés, mais les plus simples (en termes d'algorithmes) sont présentées ici.

Résultat d'affichage.

— {1, 2, 3, 4, 5, 6, 7, 8, 9}

— {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}. Les doublons ne sont pas acceptés dans un **set** !

— La fonction `pop` retire un certain élément d'un **set**. Par contre, à cause (ou grâce à) la nature des conteneurs de type **set**, on ne peut vraiment prédire quel sera l'«ordre» dans le **set**. En effet, on pourrait aussi bien obtenir `"a"` que `"b"` quand on affiche `s3.pop()`. C'est en quelque sorte une question piège ! ! **set** (de même que **dict**) fonctionne sur le principe de hashage, ce qui dépasse de très très loin les objectifs de ce guide. Pour l'affichage **print**(s3), c'est le même problème ¯_(ツ)_/¯

19 Liens importants

Le premier lien important est celui du dépôt Github du présent document. Il possède en plus, la plupart des codes qui ont servi à faire les exemples et exercices. Si vous voulez les avoir, il vous suffira donc de cloner le dépôt en allant au lien suivant :

— https://github.com/JeremieGince/ProjetPythonPhysique

20 Postface

20.1 À propos des auteurs

20.1.1 Gabriel

Vous me connaissez peut-être pour le gars qui fait des memes (du moins qui en faisait), mais je suis plus que ça ! J'ai effectué mon cégep en *Sciences informatiques et mathématiques*, un genre d'hybride entre le *DEC* en *Sciences naturelles* et le *DEC* en *Informatique*. En effet, pendant deux ans j'ai étudié la programmation orientée objet avec le langage *Java* (qui a encore sa place spéciale dans mon coeur). Je suis donc arrivé en physique avec un certain bagage informatique et c'est ce qui m'a entre autres permis d'aider d'autres personnes à programmer au courant des deux dernières années. Malheureusement, j'ai constaté que les lacunes chez certains étaient assez problématiques, c'est pourquoi j'ai pris l'initiative avec mon bon ami Gince de construire cet outil qui se veut amical et éducatif. En espérant que vous apprécierez et que cela vous aidera ! Si vous avez encore des questions ou vous voulez simplement jaser de programmation, venez me voir !

20.1.2 Gince

J'ai fait mon *DEC* en *Sciences de la nature* au cégep de Sherbrooke. Je fais présentement mon *BAC* en physique ainsi qu'un certificat en informatique. J'ai déjà travaillé avec une multitude de langages comme *Java*, *Assembleur*, *Bash*, *PHP*, *C++*, *C*, *C#*, *SQL* et surtout *python*. Je me passionne pour l'intelligence artificielle et c'est cette passion qui m'a dirigé vers python. En effet, ce langage doit être le meilleur, selon moi, pour ce domaine. Bref, si vous avez des questions ou commentaires, n'hésitez surtout pas à venir me voir.

21 À venir (version 1.0.2020)

21.1 Mises à jour

Voici les sections qui seront mises à jour :

— Modules importants

21.2 Ajouts

Voici les sections qui seront ajoutées :

— Les décorateurs

— Algèbre linéaire

— Analyse des signaux