# Module C0

Les bases de Python 3

**Timothée Duruisseau**

Department de génie mécanique
École Polytechnique Montréal
Canada
27 août 2021

# Objectifs du module

Vous trouverez dans cette documentation un condensé de la base de Python pour une utilisation en ingénierie. À la fin de cette documentation, il sera important de comprendre la base de la programmation en Python. Ainsi, il faudra savoir importer et utiliser des modules, utiliser les différents types de variables, faire les appels de fonctions et de méthodes, et utiliser les entrées et les sorties adéquatements.

# Table des matières

# Chapitre 1

# Installation

Python est un langage de programmation interprété créé au début des années 1990 par Guido van Rossum. Il s'agit d'un langage très développé, possédant plusieurs bibliothèques permettant d'accomplir presque n'importe quoi sur un ordinateur. Il est possible d'automatiser des tâches, de faire de la modélisation numérique, de récupérer et nettoyer des ensembles de données, accéder à des bases de données, envoyer des courriels de masse et bien d'autres choses encore. Dans la présente documentation, vous trouverez les ressources nécessaires pour démarrer avec Python et la programmation procédurale. Il est à noter que deux versions de Python coexiste présentement : Python 2.7 et Python 3.9. Dans le cadre du cours, la version utilisée est Python 3.9. À moins d'indication contraire, toute mention à Python dans la présente documentation fait référence exclusivement à Python 3.9.

La manière la plus simple de commencer avec Python est de procéder avec l'environnement Anaconda. Pour télécharger, cliquer sur le lien suivant : https ://www.anaconda.com/products/individual. Choisissez la version vous concernant. Si vous utilisez un ordinateur Windows, vous avez le choix entre 64-bit et 32-bit. Il est recommandé d'utiliser la version 64-bit si votre ordinateur le permet. Si vous utilisez MacOS, choisissez la version 64-Bit Graphical Installer.

## 1.1 Installation sur Windows

Une fois l'installateur téléchargé, exécutez-le en tant qu'administrateur 1.1. Procédez à l'installation.

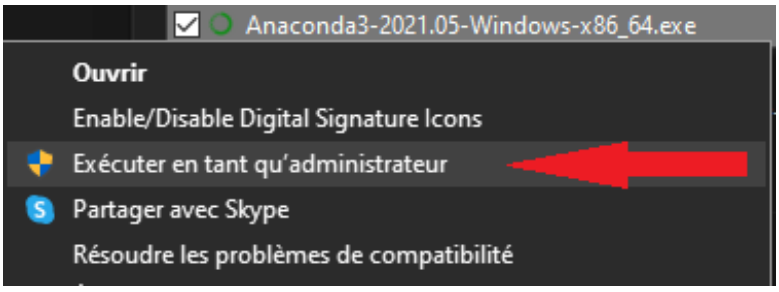


FIGURE 1.1 – Exécution en tant qu'administrateur

Il est possible que vous ayez des messages d'erreurs tel que celui de la figure 1.2. Dans ce cas, changez l'emplacement de l'installation pour une aborescence qui ne contient pas d'espace, sinon il y aura des problèmes lors de l'installation et de l'exécution de certains logiciels. Voici un exemple d'emplacement sans espace : `C:\anaconda3`.

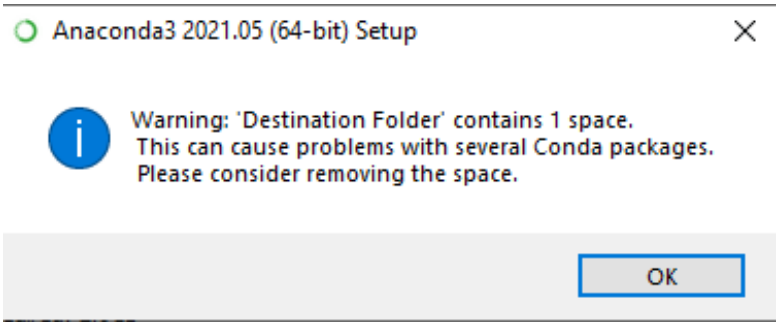


FIGURE 1.2 – Message d'erreur possible

À la fenêtre de la figure 1.3, veuillez vous assurer que les options que vous avez sont les mêmes, et cliquez ensuite sur Installer. Une fois l'installation terminée, il reste à vérifier que Spyder est bien installé dans votre distribution.

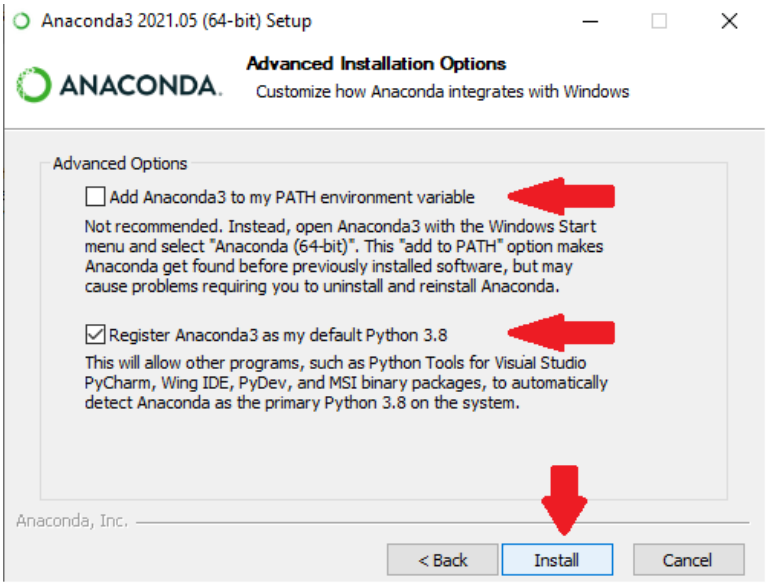


FIGURE 1.3 – Fin des options d'installation

## 1.2 Installation sur macOS

À venir

## 1.3 Alternatives

Pour les plus aguéris, il y a plusieurs alternatives pour l'utilisation de Python. En effet, puisqu'il s'agit d'un langage libre, plusieurs méthodes s'offrent à vous. Il est possible d'installer manuellement Spyder en exécution seule, installer uniquement l'interpréteur Python et éditer les fichiers *.py* directement avec Notepad++, et encore plusieurs autres variantes d'environnements de développement. Pour

le cours, uniquement Spyder avec Anaconda sera supporté, mais il y a énormément de ressources en ligne pour ceux qui souhaitent procéder autrement.

Voici quelques liens de ressources extérieures :

— Python3 : https ://www.python.org/downloads/

— Un interpréteur Python3 en ligne : https ://www.programiz.com/python-programming/online-compiler/

— Spyder : https ://www.spyder-ide.org/

— PyCharm : https ://www.jetbrains.com/fr-fr/pycharm/

— Jupyter Notebook : https ://jupyter.org/

## 1.4 Sélection du *backend* d'affichage graphique

Afin de pouvoir obtenir les figures d'une manière fiable, il est recommandé d'utilisé l'API d'affichage, *backend* en anglais, *Qt5*. Les scripts présentés ont été testé avec cet API. La sélection se fait selon la procédure suivante. Dans Spyder, sélectionnez le menu *Tools* et cliquez sur *Preferences*, tel que montré sur la figure 1.4. Ensuite, allez dans le menu *IPython Console*, *Graphics*, *Backend*, sélectionnez *Qt5* et cliquez sur *Apply*, tel que montré sur la figure 1.5.

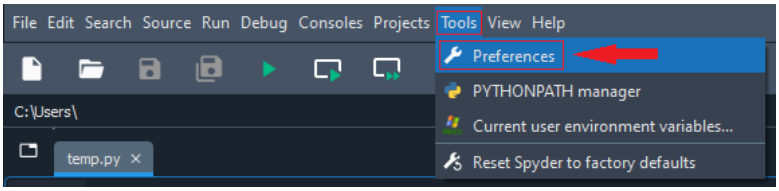


FIGURE 1.4 – Sélection des préférences

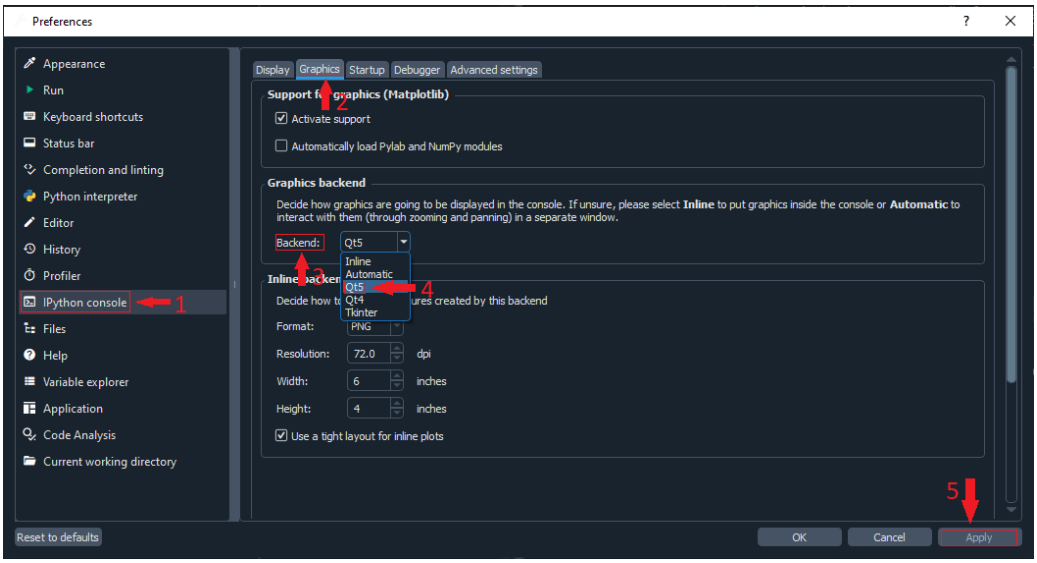


FIGURE 1.5 – Sélection de l'API *Qt5*

# Chapitre 2

# Programmation

## 2.1 La syntaxe en Python3

Python est un langage de programmation. À ce titre, comme tout langage, il nécessite une syntaxe afin d'être compréhensible. Dans cette section, vous apprendrez à écrire correctement en python.

### 2.1.1 Variables et fonctions

En programmation, il est toujours préférable de travailler avec des variables. Cela permet d'avoir de la généralité dans le code et de le rendre plus robuste.

Le script 1 est un exemple de définition de variables avec un programme simple d'addition de deux nombres.

Script 1
Exemple d'utilisation de variable

```
1 a = 2
2 b = 3
3 c = a + b
4 print('c = ' + str(c))
```

Sortie console

```
c = 5
```

On peut voir que la variable `c` est la somme de `a` et `b`. Il est à noter que lorsqu'on définie une variable, ce qui est définie est la référence à la mémoire. Cela signifie que la ligne `a = 2` signifie que `a` est une référence dans l'espace mémoire où il est inscrit la valeur numérique 2. Cela a pour implication que si on écrit une variable tel que `a = 2` et `b = a`, `b` référera le même emplacement dans la mémoire, et donc si on modifie `a`, `b` sera modifié également, et vice versa.

### 2.1.2 Mots-clés réservés

En python, il y a plusieurs mots-clés qui sont réservés à différents usages. Voici la liste.

— `False, True` : Valeurs booléennes, équivalent respectivement à 0 et 1

— `None` : Indique une allocation mémoire sans spécification. Représente la notion de rien

— `and, or, not` : Opérateurs logiques

— `as` : Permet d'affecter un alias. ie : import numpy as np

— `assert` : Permet de vérifier un code et les entrées

— `break, continue, pass` : Termes de contrôle. break permet de sortir entièrement d'une boucle. continue permet de passer à la prochaine itération. pass fait rien, donc permet de faire des blocs vides.

— `class, def` : Termes de définition. class permet de définir des classes selon les besoins. def permet de définir une fonction.

— `del` : Permet la suppression d'un objet. Cela peut être une variable, une liste, un morceau de la liste, etc

— `if, else, elif` : Termes de structure de contrôle conditionnel. if et elif doivent avoir une ou plusieurs conditions. else ne peut pas prendre de condition.

— `try, except, else, finally` : try permet de tester un bout de code sans arrêter le script au complet. except permet de définir le comportement du script si le bout de code dans le try ne fonctionne pas. else permet d'éxecuter un code uniquement si le code dans le try a fonctionné. finally exécute un code dans tous les cas.

— `for, while` : Termes permettant de déclarer des boucles. for permet une boucle dont le nombre d'itérations est connu au départ. while permet une boucle conditionnelle, c'est-à-dire que la boucle continuera de s'exécuter tant et aussi longtemps que la condition est remplie.

— `global, nonlocal` : Identifie à quel niveau la variable est définie. global permet de soit créer une variable global, soit d'appeler une variable qui n'a pas été transportée. nonlocal est basé sur le même principe, mais pour traiter des variables dans des fonctions imbriquées.

— `import, from` : Termes d'importations de modules. import permet l'importation elle-même. from permet d'importer des sections spécifiques du module

— `in` : Terme ayant deux fonctions. D'abord, il permet de vérifier si une valeur se trouve dans une séquence quelconque (list, tuple, string, etc). Ensuite, il permet de faire une itération sur les valeurs d'une séquence. Une utilisation courante de cette fonction est lors de l'écriture de la boucle for.

— `is` : Terme permettant de vérifier si deux variables réfèrent au même objet.

— `lambda` : Terme permettant la création d'une petite fonction anonyme. La fonction ne peut contenir qu'une seule expression

— `raise` : Permet de déclarer une erreur, son type et un message

— `return, yield` : Terme permettant de sortir d'une fonction et de retourner une ou des valeurs pour return, et un objet générateur pour yield

— `with` : Terme permettant de simplifier la gestion de ressources communes, tel que des fichiers par exemple

### 2.1.3 Opérateurs

Il existe en python, comme dans la plupart des langages, des opérateurs accomplissant différentes fonctions. Les opérateurs arithmétiques sont utilisés pour les opérations mathématiques. Les opérateurs d'assignation sont utilisée pour affecter une ou des valeurs à une variable. Les opérateurs comparateurs permettent de comparer deux valeurs. L'utilisation de ces opérateurs retourne True si la comparaison est vrai et False si la comparaison est fausse. Les opérateurs logiques permettent de combiner des

déclarations conditionnelles. Les opérateurs d'identité permettent de vérifier si deux objets sont le même objet ou non. Les opérateurs d'appartenance vérifient si une ou plusieurs valeurs font partie d'un objet. Finalement, les opérateurs au niveau du bit comparent des nombres binaires. Voici un lexique pour chacun des types.

| **Opérateurs** | **Fonction** | **Exemple** |
|---|---|---|
| + | Addition | `x + y` |
| - | Soustraction | `x - y` |
| * | Multiplication | `x * y` |
| / | Division | `x / y` |
| ** | Exposant | `x**y` |
| % | Modulo ; retourne le reste d'une division | `x % y` |
| // | Division entière ; retourne le quotient d'une division | `x // y` |

TABLE 2.1 – Opérateurs arithmétiques

| Opérateurs | Fonction | Exemple |
|---|---|---|
| = | Assigne une valeur à une variable | `x = 5` |
| += | Incrémente une variable de la valeur indiquée | `x += 3` |
| -= | Décrémente une variable de la valeur indiquée | `x -= 3` |
| *= | Multiplie la valeur de la variable par la valeur indiquée | `x *= 3` |
| /= | Divise la valeur de la variable par la valeur indiquée | `x /= 3` |
| **= | Puissance de la variable par la valeur indiquée | `x **= 3` |
| //= | Division entière de la variable par la valeur indiquée | `x //= 3` |
| %= | Assigne le modulo de la variable par la valeur indiquée | `x %=3` |
| &= | Assigne la valeur suite à une comparaison d'intersection binaire (AND) | `x &= 3` |
| \|= | Assigne la valeur suite à une comparaison d'inclusion binaire (OR) | `x \|=3` |
| ^ = | Assigne la valeur suite à une comparaison d'exclusion binaire (XOR) | `x ^= 3` |
| «= | Assigne la valeur suite à un décalage des bits vers la gauche | `x «= 3` |
| »= | Assigne la valeur suite à un décalage des bits vers la droite | `x »= 3` |

TABLE 2.2 – Opérateurs d'assignation

| Opérateurs | Fonction | Exemple |
|---|---|---|
| == | Égale | `x == y` |
| != | Différent | `x != y` |
| > | Strictement plus grand que | `x > y` |
| < | Strictement plus petit que | `x < y` |
| >= | Plus grand ou égale | `x >= y` |
| <= | Plus petit ou égale | `x <= y` |

TABLE 2.3 – Opérateurs de comparaison

| Opérateurs | Description | Exemple |
|---|---|---|
| and | Retourne True si toutes les déclarations sont vraies | `x < 5 and x < 10` |
| or | Retourne True si au moins une déclaration est vraie | `x < 5 or x > 4` |
| not | Inverse le résultat obtenu ; True devient False, False devient True | `not(x < 5 and x < 10)` |

TABLE 2.4 – Opérateurs logiques

| Opérateurs | Description | Exemple |
|---|---|---|
| is | Retourne True si les deux variables sont le même objet | `x is y` |
| is not | Retourne True si les deux variables ne sont pas le même objet | `x is not y` |

Table 2.5 – Opérateurs d'identité

| Opérateurs | Description | Exemple |
|---|---|---|
| in | Retourne True si la séquence de valeurs spécifiées est présente dans l'objet | `x in y` |
| not in | Retourne True si la séquence de valeurs spécifiées n'est pas présente dans l'objet | `x not in y` |

Table 2.6 – Opérateurs d'appartenance

| Opérateurs | Nom | Description |
|---|---|---|
| & | AND | Pose la valeur à 1 si les deux bits comparés sont 1 |
| \| | OR | Pose la valeur à 1 si au moins un des deux bits est 1 |
| ^ | XOR | Pose la valeur à 1 si un seul bit est égale à 1 |
| ~ | NOT | Inverse tous les bits |
| « | Décalage à gauche | Décale tous les bits vers la gauche en insérant des 0 à droite |
| » | Décalage à droite | Décale tous les bits vers la droit en insérant des 0 à gauche |

Table 2.7 – Opérateurs au niveau du bit

## 2.2 Types de données

Dans cette section, nous explorerons les types de données utilisés avec Python. Les types déterminent les opérations possibles de faire ainsi que la manière dont l'ordinateur et l'interpréteur doivent considérer les différents objets. Nous commencerons par voir les types natifs à Python. Ensuite, nous verrons les types associés aux modules qui seront utilisés durant le cours.

### 2.2.1 Types natifs

#### Les nombres : `int, float, complex`

Les nombres ont trois types distincts : entiers, flottants et complexes. Les nombres entiers, *integer*, sont représentés par un numbre tout simplement (ie : 3). Les nombres flottants, *float*, sont représentés avec un point, (ie : 3.1416 ou 3.). Les nombres complexes, *complex*, sont représentés par une partie réelle, plus la partie imaginaire (ie : 3 + 4j). La partie imaginaire est noté avec un j.

Script 2
Différents types de nombres

```
1 a = 5
2 b = 5.
3 c = 5.4
4 d = 5 + 4j
5
6 print('La variable a est de type '+ str(type(a)))
7 print('La variable b est de type '+ str(type(b)))
8 print('La variable c est de type '+ str(type(c)))
9 print('La variable d est de type '+ str(type(d)))
```

Sortie console

```
La variable a est de type <class 'int'>
La variable b est de type <class 'float'>
La variable c est de type <class 'float'>
La variable d est de type <class 'complex'>
```

#### Les chaînes de caractères : `str`

Les chaînes de caractères, ou *string* en anglais, sont des séquences de bytes représentants des caractères Unicode. Puisque c'est une séquence, il est possible d'accéder à certains caractères spécifiquement en indexant. Les chaînes de caractères sont identifiés grâce à des guillemets. Les guillemets peuvent être simples, doubles, ou triples. Voici quelques exemples de chaînes de caractères et les méthodes pour les créer et les accéder.

À noter que l'indexation peut se faire également à rebours, où -1 représente le dernier caractère, -2 l'avant-dernier caractère, et ainsi de suite.

#### Les listes et les tuples : `list, tuple`

Les listes et les tuples sont également des séquences de bytes comme les chaînes de caractères. Cela dit, ils ne sont pas limités à des caractères Unicode. Les listes et les tuples peuvent contenir n'importe

SCRIPT 3
Exemple de manipulation et d'indexation de chaîne de caractères

```
1  str1 = 'Bienvenue en MEC1315'
2  print(str1)
3
4  str2 = "Démontrons l'utilisation des guillemets doubles"
5  print(str2)
6
7  str3 = str2[0:13]
8  print(str3 + "indexation")
9
10 str4 = "Il est également possible \n d'écrire sur plusieurs lignes \n en faisant des
       sauts de ligne"
11 print(str4)
```

Sortie console

```
Bienvenue en MEC1315
Démontrons l'utilisation des guillemets doubles
Démontrons l'indexation
Il est également possible
 d'écrire sur plusieurs lignes
 en faisant des sauts de ligne
```

quel type de données. Il est donc possible de faire des structures de données très complexes pour faire le transport. Bien que les listes et les tuples soient très similaires, ils ont une grande différence. Les listes sont muables, c'est-à-dire que le contenu peut être changé. Les tuples sont immuables, ce qui veut dire que le contenu ne peut être changé. Pour changer un élément dans un tuple, il faut changer l'objet au complet.

Les listes sont déclarées en utilisant des crochets. Les tuples sont déclarés en utilisant des parenthèses.

L'indexation dans les tuples et les listes est séquencielle.

#### Les dictionnaires : `dict`

Les dictionnaires sont des structures de données dont les valeurs sont stockées en pairs clé :valeur. Une clé peut être n'importe quel type de données immuable. Une valeur peut être n'importe quel type de données ou de structures de données.

### 2.2.2 Types numpy

Le module numpy possède ses propres types pour faire du calcul scientifique. Ceux-ci permettent de faire des calculs matriciels et de résoudre des systèmes d'équations. Dû à la nature du module, plusieurs types sont redondants avec les types natifs. Cependant, certains seront tous de même expliqués. Afin de simplifier l'écriture, les types qui normalement s'écriraient `numpy.*` seront écrits `np.*`. Les explications de cette simplification sont présentes dans la section d'importation.

SCRIPT 4
Démonstration des listes et des tuples

```
list1 = [1,2,3]
list2 = [[1,2,3],['p','t'],['r',4,'rapport',5]]
list3 = [(8,7),[2,3,4],{'notes':[6,6,7,6]}]

tuple1 = (1,2,3)
tuple2 = ((1,2,3),('travail','r'))
tuple3 = ([8,7],(2,3,4),{'retour':(18,19,20)})

print('\nLa liste list1 est de type ' + str(type(list1)))
print('\nLa liste list2 est de type ' + str(type(list2)))
print('\nLa liste list3 est de type ' + str(type(list3)))

print('\nLa liste tuple1 est de type ' + str(type(tuple1)))
print('\nLa liste tuple2 est de type ' + str(type(tuple2)))
print('\nLa liste tuple3 est de type ' + str(type(tuple3)))
```

Sortie console

```
La liste list1 est de type <class 'list'>

La liste list2 est de type <class 'list'>

La liste list3 est de type <class 'list'>

La liste tuple1 est de type <class 'tuple'>

La liste tuple2 est de type <class 'tuple'>

La liste tuple3 est de type <class 'tuple'>
```

SCRIPT 5
Exemple de dictionnaire

```
dict1 = {'l':[2,3],3:'jacques',(9,3):{'h':3}}

print(dict1['l']) # Exemple d'indexation par clé de type string
print(dict1[(9,3)]) # Exemple d'indexation par clé de type tuple
```

Sortie console

```
[2, 3]
{'h': 3}
```

**Nombres : `np.int32, np.float64`**

Il existe plusieurs types de nombre dans le module `numpy`. Ils ont tous une utilité dans un domaine ou un autre. Cependant, seuls les plus utilisés seront présentés ici.

Les entiers sont identifiés par soit `np.int` ou `np.int32`. La différence entre les deux est que la mémoire allouée pour `np.int` est définie par la plateforme utilisée alors que `np.int32` a toujours la même mémoire, permettant les nombres de -2147483648 à 2147483647. Le type `np.float64` permet de stocker des nombres à virgule avec une double précision.

Ces types, bien qu'ils peuvent être utilisés dans les mêmes contextes que les équivalents natifs de python, apparaissent surtout lors de la création et manipulation de matrices et de vecteurs.

**Matrices et vecteurs : `np.ndarray`**

Les matrices et les vecteurs sont regroupés à l'intérieur d'un même type : `np.ndarray`. Un vecteur est un ensemble de données sans dimension. Cela implique que le vecteur est autant une ligne qu'une colonne lors des opérations. Une matrice est un ensemble de données ayant 2 dimensions. La création manuelle d'une matrice se fait ligne par ligne. De plus, lorsque les dimensions de la matrice sont requises, il faut faire attention à ce que l'on cherche à obtenir. Portez une attention particulière aux exemples ci-dessous pour voir la différence entre les différentes méthodes. Voici des exemples de création et de manipulation de vecteur et matrice.

Il est possible d'accomplir des opérations vectorielles sur les matrices et les vecteurs. Par défaut, les opérations sont faites éléments par éléments. Donc l'addition de deux vecteurs ensemble retournera un vecteur dont la valeur à l'indice 0 sera la somme des deux valeurs d'indice 0, et ainsi de suite. Même chose pour la soustraction, multiplication, et la division.

SCRIPT 6
Démonstration de matrices et vecteurs avec `numpy`

```
vec1 = np.array([0,1,2,3,4])
print('Le vecteur vec1 est de type ' + str(type(vec1)))

vec2 = np.zeros(4) # Création d'un vecteur de zéros
print('Le vecteur vec2 est de type ' + str(type(vec2)))

mat1 = np.array([[0,1,2],[3,4,5],[6,7,8]])
print('La matrice mat1 est de type ' + str(type(mat1)))
print('La forme de la matrice mat1 est : ' + str(mat1.shape))  # Retourne un tuple du
    format (nombre de lignes, nombre de colonnes)
print('La matrice mat1 contient %d lignes' % len(mat1))   # Retourne la grandeur de la
    dimension 0, soit le nombre de lignes
print('La matrice mat1 contient %d colonnes.' % len(mat1[0]))  # Retourne la grandeur
    de la dimension 1, soit le nombre de colonnes

mat2 = np.ones([3,4]) # Création d'une matrice de 1
print('La matrice mat2 est de type ' + str(type(mat2)))
print('La forme de la matrice mat2 est : ' + str(mat2.shape))  # Retourne un tuple du
    format (nombre de lignes, nombre de colonnes)
print('La matrice mat2 contient %d lignes' % len(mat2))   # Retourne la grandeur de la
    dimension 0, soit le nombre de lignes
print('La matrice mat2 contient %d colonnes.' % len(mat2[0]))  # Retourne la grandeur
    de la dimension 1, soit le nombre de colonnes
```

Sortie console

```
Le vecteur vec1 est de type <class 'numpy.ndarray'>
Le vecteur vec2 est de type <class 'numpy.ndarray'>
La matrice mat1 est de type <class 'numpy.ndarray'>
La forme de la matrice mat1 est : (3, 3)
La matrice mat1 contient 3 lignes
La matrice mat1 contient 3 colonnes.
La matrice mat2 est de type <class 'numpy.ndarray'>
La forme de la matrice mat2 est : (3, 4)
La matrice mat2 contient 3 lignes
La matrice mat2 contient 4 colonnes.
```

## 2.3 Structures de contrôle : `if`, `elif`, `else`

En programmant, il est important de pouvoir contrôler le comportement du logiciel selon les informations que le logiciel possède. Pour faire ce contrôle, on utilise les mots-clés `if`, `elif` et `else`.

Le mot-clé `if` est toujours le premier à être utiliser pour un bloc. À la suite du mot-clé, on inscrit le test qui doit être fait. Par défaut, `if` procède avec le code appartenant à son bloc uniquement si le test retourne `True`. Une fois le bloc de code exécuté, l'interpréteur sort de la structure courante et passe à la prochaine ligne de code. Si le test retourne `False`, alors les mots-clés `elif` et `else` peuvent avoir leur code respectif exécuté.

Le mot-clé `elif` indique que si le test précédent retourne `False`, alors un autre test doit être exécuté. Si celui-ci retourne `True`, le code du bloc est exécuté et sort ensuite de la structure et l'interpréteur poursuit avec la prochaine ligne de code. Si le test retourne `False`, le bloc n'est pas exécuté et l'interpréteur passe à la prochaine ligne.

Le mot-clé `else` est la dernière partie d'une structure de contrôle. Si tous les tests précédents ont retourné `False`, alors le code du bloc est exécuté et l'interpréteur sort de la structure. Ce mot-clé ne fait aucun test et donc le code sera toujours exécuté si les tests en `if` et `elif` retournent tous `False`.

Dans les structures de contrôle, les espaces blancs doivent être respectés. Le code appartenant au bloc `if` est tabulé par rapport au mot-clé. Les mots-clés `elif` et `else` ne sont pas tabulé par rapport au `if` auquels ils sont reliés, mais leur code respectif est tabulé.

Il est important de savoir que l'entrée dans les blocs se font avec les deux-points.

Les structures peuvent imbriquées, selon ce qui est nécessaire pour avoir le comportement souhaité. Encore une fois, les espaces blancs doivent être respectés afin que le code s'exécute tel que souhaité.

Script 7
Démonstration d'une structure de contrôle simple

```
1 a = 5
2
3 # Structure de contrôle. Remarquer la présence des deux-points
4 if a == 4:
5     print('Le test exécuté a retourné True.')
6 else:
7     print('Le test exécuté a retourné False.')
```

Sortie console

```
Le test exécuté a retourné False.
```

SCRIPT 8
Démonstration d'une structure de contrôle simple avec un deuxième test

```
a = 5

# Structure de contrôle. Remarquer la présence des deux-points
if a == 4:
    print('Le premier test exécuté a retourné True.')
elif a <= 6:
    print('Le deuxième test exécuté a retourné True.')
else:
    print('Tous les tests exécutés ont retourné False.')
```

Sortie console

```
Le deuxième test exécuté a retourné False.
```

SCRIPT 9
Démonstration de structures de contrôle imbriquées

```
a = 5
b = 'mode'

# Premier niveau de structure, remarquez les espaces blancs
if a > 4:
    print('Le test du premier niveau a retournée True.')
    # Deuxième niveau de structure
    if type(b) is str:
        print('Le premier test du deuxième niveau a retourné True\n et la variable b est
            une chaîne de caractère.')
    elif type(b) is float:
        print("Le premier test du deuxième niveau a retourné False,\n le deuxième test
            du deuxième niveau a retournée True\n et la variable b est un nombre
            flottant.")
else:
    print('Le test du premier niveau exécuté a retournée False.')
```

Sortie console

```
Le test du premier niveau a retournée True.
Le premier test du deuxième niveau a retourné True
 et la variable b est une chaîne de caractère.
```

## 2.4 Les boucles itératives

Très souvent, il est nécessaire de procéder à plusieurs itérations dans l'exécution du code. Pour faire cela, il existe deux types de boucles. Le premier type est la boucle dont le nombre d'itérations est déterminé en avance. Le mot-clé associé à ces boucles est `for`. Le deuxième type est la boucle dont le nombre d'itérations est indéterminé et l'entrée dans la boucle se fait par l'exécution d'un test qui doit retourner `True`. Si le test retourne `False`, la boucle est arrêtée. Tout comme les structures de contrôle, l'entrée dans les boucles se fait par les deux-points et les espaces blancs doivent être respectés. Les boucles de tous types peuvent être imbriquées, au besoin.

### 2.4.1 Boucles `for`

Sous sa forme la plus simple, la boucle `for` demande un nom de variable pour l'itération et un objet sur lequel l'itération sera faîte. L'objet peut être une liste ou un tuple d'entiers, avec la fonction `range()` étant couramment utilisé pour généré l'objet, prenant en argument le nombre d'itération souhaitée. Il est à noter que Python considère le début d'une séquence à 0, et donc une liste séquentielle de 10 éléments est construite comme suit : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Aussi, l'objet itéré n'a pas besoin d'avoir des nombres entiers qui se suivent. La boucle passera à travers l'objet de manière séquentielle. À ce titre, uniquement les objets itérables peuvent être utilisés.

Script 10
Démonstration d'une boucle `for` simple

```
1 # Boucle for. Remarquez la syntaxe, utilisant les mots-clés for et in
2 for i in range(5):
3     print('i = %d' % i)
```

Sortie console

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Bien souvent, il est également nécessaire d'itérer sur un objet mais deux informations sont nécessaires : l'indice et la valeur. Bien qu'il existe des méthodes pour procéder à ce genre d'itération, Python fournit une méthode très directe avec la fonction `enumerate(objet)`. La fonction retourne deux objets itérables, un contenant les indices et l'autre les valeurs de l'objet entré en argument.

### 2.4.2 Boucles `while`

Les boucles `while` permettent de faire des opérations tant et aussi longtemps qu'une condition est remplie. Sous sa forme la plus simple, la boucle est démarrée avec le mot-clé et le test à faire pour entrer dans la boucle. Ce test est fait au début de chaque itération. Il est important de s'assurer que la boucle `while` puisse se terminer. En effet, puisque la boucle fait un test à chaque itération, si la condition est toujours remplie, on se trouve avec une boucle infinie, qui empêche le reste du code de s'exécuter pour les cas bénins et peut faire des dommages irréparables sur la machine dans les cas les plus extrêmes.

SCRIPT 11
Démonstration d'une boucle `for` avec la fonction `enumerate()`

```
a = [‘a’,‘b’,‘d’,‘e’]
# Boucle for avec enumerate
for i,val in enumerate(a):
    print("La valeur à l’indice %d est %s" % (i,val))
```

Sortie console

```
La valeur à l’indice 0 est a
La valeur à l’indice 1 est b
La valeur à l’indice 2 est c
La valeur à l’indice 3 est d
```

SCRIPT 12
Démonstration d'une boucle `while` simple

```
i = 2

# Boucle while avec test
while i < 7:
    print(’La valeur de i est %d.’ % i)
    # Incrément de i pour s’assurer de ne pas avoir une boucle infinie
    i += 1

print(’La valeur final de i est %d’ % i)
```

Sortie console

```
La valeur de i est 2.
La valeur de i est 3.
La valeur de i est 4.
La valeur de i est 5.
La valeur de i est 6.
La valeur final de i est 7
```

## 2.5 Les structures d'essai : `try`

Parfois, il est pertinent d'exécuter un morceau code, sans que cela ne fasse planter le code au complet. Pour faire cela, on fait appel aux structures d'essai, utilisant le mot-clé `try`. Cela dit à l'interpréteur que si ce qui est dans le bloc retourne une erreur, de ne pas prendre compte de l'erreur, de sortir du bloc et de continuer avec la prochaine ligne de code.

Toutes les structures d'essai nécessitent au minimum les mots-clés `try` et `except`. Ce deuxième mot-clé indique à l'interpréteur quoi faire si le code rencontre une erreur. Avec le mot-clé `except`, il est possible de spécifier différentes actions selon l'erreur obtenue. Il faut donc spécifier le type d'erreur à la suite du mot-clé, par exemple `except NameError:`. Il est de bonne pratique qu'à la suite des erreurs qui sont spécifiées, de rajouter un autre bloc `except` qui prendra en compte toutes les autres erreurs qui n'ont pas été prise en compte. En effet, si rien ne suit le mot-clé d'exception, toutes les erreurs sont considérées.

De plus, il est également possible d'exécuter du code selon que le bloc `try` a fonctionné sans erreur, avec le mot-clé `else`, ou qui soit exécuté après l'essai et les exceptions, avec le mot-clé `finally`.

Script 13
Démonstration d'une structure d'essai simple

```
1  a = 'Bonjour '
2  b = 'le monde !'
3
4  try:
5      # Concaténation de deux chaîne de caractères
6      c = a + b
7      print("Action dans la structure d'essai réussie.")
8      print('Résultat : %s' % c)
9  except:
10     print('Le code a généré une erreur, donc le code du EXCEPT est exécuté.')
```

Sortie console

```
Action dans la structure d'essai réussie.
Résultat : Bonjour le monde !
```

SCRIPT 14
Démonstration d'une structure d'essai avec une exception spécifique

```
a = 'Bonjour '
b = 'le monde !'

try:
    # Concaténation de deux chaîne de caractères
    c = a - b
    print("Action dans la structure d'essai réussie.")
    print('Résultat : %s' % c)
except TypeError:
    print("Le code en essai ne fonctionne pas car il retourne une erreur TypeError.")
except:
    print("Une erreur est survenue qui n'est pas une erreur TypeError.")
```

Sortie console

```
Le code en essai ne fonctionne pas car il retourne une erreur TypeError.
```

SCRIPT 15
Démonstration d'une structure d'essai avec un code exécuté si et seulement si l'essai est réussi

```
a = 'Bonjour '
b = 'le monde !'

try:
    # Concaténation de deux chaîne de caractères
    c = a + b
    print("Action dans la structure d'essai réussie.")
    print('Résultat : %s' % c)
except:
    print("Une erreur est survenue.")
else:
    print("Le code en essai est réussi et donc ce message s'affiche.")
```

Sortie console

```
Action dans la structure d'essai réussie.
Résultat : Bonjour le monde !
Le code en essai est réussi et donc ce message s'affiche.
```

SCRIPT 16
Démonstration d'une structure d'essai avec un code exécuté à la fin du bloc, peu importe le succès du code

```
a = 'Bonjour '
b = 'le monde !'

try:
    # Concaténation de deux chaîne de caractères
    c = a - b
    print("Action dans la structure d'essai réussie.")
    print('Résultat : %s' % c)
except:
    print("Une erreur est survenue.")
else:
    print("Le code en essai est réussi et donc ce message s'affiche.")
finally:
    print("Ce message s'affiche peu importe le succès du code en essai.")
```

Sortie console

```
Une erreur est survenue.
Ce message s'affiche peu importe le succès du code en essai.
```

## 2.6 Les modules

Une des forces de Python est sa capacité à étendre les possibilités en utilisant des librairies adaptés aux besoins. Ces librairies sont couramment appelés des modules, ou *packages* en anglais. Pour les utiliser, il faut les importer explicitement dans le code. Dans cette section, les méthodes d'importation seront présentés et ensuite, plusieurs modules importants seront présentés.

### 2.6.1 Importation des modules

Il existe plusieurs façons d'importer un module en Python, mais toutes ces façons demandent l'utilisation du mot-clé `import`.

La méthode d'importation la plus simple consiste à uniquement écrire `import nom_du_module` dans l'entête du script. Ensuite, pour utiliser les fonctions et méthodes de ce module, il faut ensuite écrire `nom_du_module.fonction()`. Évidemment, lorsque ce module est très utilisé dans un script, il peut devenir rapidement lassant d'écrire le nom du module au long pour chaque fonction utilisée. Il y a deux manières de remédier à ce problème.

La première manière consiste à donner un alias au nom du module, en inscrivant par exemple `import nom_du_module as alias`. Cela permet d'ensuite appeler les fonctions du module en écrivant `alias.fonction()`. L'alias peut être n'importe quel amalgame de lettres et chiffres, tant et aussi longtemps qu'il ne s'agit pas de mot-clé et qu'il n'y a pas de caractères spéciaux autres que la barre de soulignement. Cette manière est préférée à la seconde manière car cela évite des problèmes lorsqu'on importe plusieurs modules en même temps.

La deuxième manière consiste à importer le module dans son ensemble et de le mettre dans le même espace de noms, *namespace*, que les fonctions natives de python en écrivant `from nom_du_module import *`. En utilisant cette manière, il n'est plus nécessaire d'écrire le nom du module en préfixe à

la fonction, il suffit d'appeler la fonction directement. De manière générale, il est déconseillé d'utiliser cette manière car il peut y avoir des conflits entre plusieurs modules. Par exemple, 2 modules peuvent avoir le même nom pour une fonction, et ainsi cela devient ambigu pour l'interpréteur.

Par ailleurs, il peut arriver qu'un module ne soit pas nécessaire dans son ensemble. Dans ce cas, il est possible d'importer uniquement les parties nécessaires, en écrivant par exemple `from nom_du_module import fonction1, fonction2, fonction3`. On peut importer une seule fonction tout comme on peut importer plusieurs fonctions du même module en séparant les noms par des virgules. Pour utiliser les fonctions, il suffit d'appeler la fonction par son nom, sans préfixe.

### 2.6.2 Module `numpy`

Un module très utilisé en calcul scientifique est `numpy`. Ce module permet l'utilisation de vecteurs et de matrices, ainsi que différentes fonctions mathématiques telles que les fonctions trigonométriques et logarithmiques. Il est également possible d'aller chercher certaines constantes mathématiques comme $\pi$ et $e$. Il est important de noter que lors de l'importation du module, il est d'usage d'affecter l'alias `np` au module `numpy`. Bien que n'importe quel alias valide fonctionnera, l'utilisation de cette norme permettra une meilleure harmonisation pour de grands projets avec plusieurs collaborateurs.

Le tableau 2.8 présente quelques fonctions très utilisées dans ce module.

#### Algèbre linéaire

Le module `numpy` permet aussi de faire de l'algèbre linéaire. Le calcul matriciel et vectoriel permet de traiter énormément de données sans passer par des boucles, ce qui permet d'accélérer le code. Cela peut être la différence entre un code qui roule en quelques secondes et un code qui roule en quelques minutes. L'appel des fonctions d'algèbre linéaire nécessitent le préfixe `np.linalg`. Dans le tableau 2.9, il y a les fonctions qui seront utiles pour énormément d'applications.

| | |
|---|---|
| `np.pi` | $\pi$ |
| `np.e` | Nombre d'euler $e$ |
| `np.array(objet,dtype=None)` | Crée un vecteur ou une matrice. `objet` est la matrice ou le vecteur, obligatoire. `dtype` définie le type des valeurs de la matrice ou du vecteur, optionnel. |
| `np.asarray(a,dtype=None)` | Convertie l'entrée en matrice ou vecteur. L'entrée peut être n'importe quelle combinaison de listes, de tuples et de vecteurs. Noter que les grandeurs doivent concorder. |
| `np.empty(forme,dtype=float)` | Crée un vecteur ou une matrice vide. `forme` peut être un entier, une liste d'entiers, un tuple d'entiers ou un vecteur d'entiers, obligatoire. `dtype` définie le type des valeurs de la matrice ou du vecteur, optionnel. |
| `np.zeros(forme, dtype=float)` | Crée un vecteur ou une matrice de 0. `forme` peut être un entier, une liste d'entiers, un tuple d'entiers ou un vecteur d'entiers, obligatoire. `dtype` définie le type des valeurs de la matrice ou du vecteur, optionnel. |
| `np.ones(forme, dtype=float)` | Crée un vecteur ou une matrice de 1. `forme` peut être un entier, une liste d'entiers, un tuple d'entiers ou un vecteur d'entiers, obligatoire. `dtype` définie le type des valeurs de la matrice ou du vecteur, optionnel. |
| `np.eye(N,M,k=0,dtype=float)` | Crée une matrice identitée. `N` est le nombre de lignes, obligatoire. `M` est le nombre de colonnes ; par défaut, la valeur est identique à N, optionnel. `k` est l'indice de la diagonale ; 0 est la diagonale principale, les valeurs positives réfèrent aux diagonales supérieures, les valeurs négatives réfèrent aux diagonales inférieures. `dtype` définie le type des valeurs de la matrice ou du vecteur, optionnel. |
| `np.linspace(debut,fin,num=50)` | Crée un vecteur linéaire à pas égal. `debut` est le premier élément ; `fin` est le dernier élément. `num` est le nombre d'éléments du vecteur, valeur par défaut de 50. |
| `np.arange(debut,fin,step=1)` | Crée un vecteur à pas égal. `debut` est le premier élément, optionnel ; `fin` est la limite de l'intervalle, obligqatoire. `step` est le pas, valeur par défaut de 1. |
| `np.sum(a)` | Retourne la somme des éléments du vecteur ou de la matrice. |
| `np.diag(v,k=0)` | Extrait une diagonale ou construit une matrice diagonale. Pour l'extraction, `v` est une matrice. Pour la construction, `v` est un vecteur. `k` est l'index de la diagonale. |
| `np.sin(a)` | Fonction trigonométrique sinus |
| `np.cos(a)` | Fonction trigonométrique cosinus |
| `np.tan(a)` | Fonction trigonométrique tangeante |
| `np.exp(a)` | Fonction exponentielle |
| `np.log(a)` | Fonction logarithme naturelle |

TABLE 2.8 – Tableau des fonctions courantes de `numpy`

| | |
|---|---|
| `np.dot(a,b)` | Retourne le produit scalaire ou le produit matricielle, en fonction de ce qui est donné en arguments |
| `np.vdot(a,b)` | Retourne le produit scalaire de deux vecteurs |
| `np.linalg.eig(a)` | Retourne les valeurs propres et les vecteurs propres d'une matrice carrée |
| `np.linalg.eigvals(a)` | Retourne les valeurs propres d'une matrice quelconque |
| `np.linalg.norm(x)` | Retourne la norme d'un vecteur ou d'une matrice |
| `np.linalg.det(a)` | Retourne le déterminant d'une matrice |
| `np.linalg.trace(a)` | Retourne la trace d'une matrice |
| `np.linalg.solve(a,b)` | Résout un système d'équation |
| `np.linalg.inv(a)` | Retourne l'inverse d'une matrice |
| `np.linalg.pinv(a)` | Retourne la pseudo-inverse (Moore-Penrose) d'une matrice |

TABLE 2.9 – Tableau de fonctions d'algèbre linéaire de `numpy`

### 2.6.3 Module `matplotlib`

Ce module est principalement utilisé pour l'affichage de graphiques. Dans son entiereté, c'est un module très complet, permettant un contrôle total de l'affichage. Cela dit, certaines collections de fonctions sont plus utilisées que d'autres. Notamment, il y a `pyplot`. Cette collection de fonctions est couramment importé en utilisant l'alias `plt`, en utilisant la commande `import matplotlib.pyplot as plt`. Cet alias est bien entendu afin de raccourcir l'écriture car l'inscription du préfixe complet peut rapidement devenir agaçant.

| | |
|---|---|
| `plt.subplot()` | Permet l'affichage de plusieurs graphiques sur une même figure, en utilisant la méthode pyplot |
| `plt.subplots()` | Permet l'affichage de plusieurs graphiques sur une même figure, en utilisant la méthode orientée objet |
| `plt.figure()` | Permet la création d'une nouvelle figure ou l'activation d'une figure dejà existante |
| `plt.plot()` | Trace un graphique linéaire |
| `plt.scatter()` | Trace un nuage de points |
| `plt.xlabel()` | Identifie l'axe $x$ avec la chaîne de caractères donnée en argument ; utilisation plus fréquente avec la méthode pyplot |
| `plt.set_xlabel()` | Identifie l'axe $x$ avec la chaîne de caractères donnée en argument ; utilisation plus fréquente avec la méthode orientée objet |
| `plt.ylabel()` | Identifie l'axe $y$ avec la chaîne de caractères donnée en argument ; utilisation plus fréquente avec la méthode pyplot |
| `plt.set_ylabel()` | Identifie l'axe $y$ avec la chaîne de caractères donnée en argument ; utilisation plus fréquente avec la méthode orientée objet |
| `plt.title()` | Place un titre pour le graphique actif avec la chaîne de caractères donnée en entrée ; utilisation plus fréquente avec la méthode pyplot |
| `plt.set_title` | Place un titre pour le graphique actif avec la chaîne de caractères donnée en entrée ; utilisation plus fréquente avec la méthode orientée objet |

TABLE 2.10 – Tableau des fonctions courantes de `matplotlib`

# Chapitre 3

# Les entrées et les sorties

Lors de la cération d'un logiciel, il est pertinent de faire en sorte que les données à traiter ne soient pas codé en dur. Pour ce faire, il est possible d'entrer des données pendant l'exécution du logiciel en utilisant des fonctions d'entrées, *input* en anglais, ou en important des fichiers contenant les données que l'on souhaite traiter. Il est également pertinent de pouvoir observer les données et les résultats. Dans ce chapitre, nous verrons les entrées consoles et les entrées graphiques, la lecture de fichier, l'écriture dans la console et dans les fichiers et finalement, les différents graphiques qui peuvent être générés pour visualiser les données et leur traitements.

## 3.1 Entrées consoles

Ce qu'on appelle les entrées consoles sont des entrées faites à travers la console Python. Ainsi, la manière la plus simple d'acquérir une entrée d'utilisateur est d'utiliser `input()`. Cette fonction est native à Python et ne nécessite donc pas de préfixe. Cette fonction retourne toujours une chaîne de caractères. Cette chaîne de caractères peut ensuite être transformé dans le type que l'on souhaite utiliser. Voici quelques exemples d'utilisation.

Bien sûr, lors de la transformation de la chaîne de caractères dans un autre type de données, il est possible d'obtenir des erreurs si l'entrée n'a pas bien été faite. Par exemple, si en entrée on s'attend à un nombre entier et que l'usager entre une chaîne de caractères ou un nombre à décimal, l'interpréteur retournera une erreur car les types ne concordent pas. Pour remédier à ce problème, il est possible de faire de la gestion d'erreurs, soit avec des structures conditionnelles, soit avec des blocs `try ... except`. Les blocs `try ... except` sont privilégiés pour ce genre de problème car la gestion des erreurs est beaucoup plus simple et exhaustive, tout en réduisant la taille du code. Les explications concernant les blocs `try ... except` se trouvent ici.

Il est également possible d'entrer plusieurs valeurs dans la même ligne et même plusieurs valeurs sur plusieurs lignes. Voici 2 exemples pour plusieurs valeurs sur une même ligne.

La méthode `split()` permet de casser la chaîne de caractères pour faire une liste quelconque. Cette méthode peut prendre deux paramètres facultatifs. Le premier est `separator`, qui permet de déterminer à quel endroit la chaîne est cassée ; par défaut, il y a séparation pour chaque espace blanc. Le second paramètre est `maxsplit`, qui permet de déterminer combien de cassures qui pourront être faites ; par défaut, la valeur est de -1, ce qui signifie qu'il y aura autant de cassures que de séparateur.

La fonction `map(fonction,iterable)` permet d'appliquer une fonction à tous les éléments d'un objet itérable. Ainsi, dans l'exemple précédent, la fonction `map()` applique la fonction `int()` à tous les éléments de la liste `liste_entree` pour en faire une liste d'entiers.

Voici maintenant un exemple pour obtenir des entrées de valeurs sur plusieurs lignes.

SCRIPT 17

```
nom = input('Quel est votre nom ? ')
print('Bonjour ' + nom + '!')

nb_cours = int(input('Combien de cours avez-vous ? '))
print('Vous avez %d cours par semaine.' % nb_cours)

num1 = float(input('Entrez un premier nombre à décimal : '))
num2 = float(input('Entrez un deuxième nombre à décimal : '))
resultat = num1 * num2
print('Le produit des deux nombres que vous avez entré est %.6f' % resultat)
```

```
Quel est votre nom ? Jean
Bonjour Jean!

Combien de cours avez-vous ? 6
Vous avez 6 cours par semaine.

Entrez un premier nombre à décimal : 3.4

Entrez un deuxième nombre à décimal : 5.8
Le produit des deux nombres que vous avez entré est 19.720000
```

SCRIPT 18
Plusieurs valeurs sur la même ligne

```
1  nom, age, score = input("Entrez le nom, l'âge et le score de l'étudiant, séparés par
       des espaces : ").split()
2  print('Nom : ', nom)
3  print('Âge : ', age)
4  print('Score : ', score)
5
6  liste_entree = input("Entrez une liste de nombres séparés par des espaces : ").split()
7  print("Liste obtenue par input : ", liste_entree)
8
9  liste_nombre = list(map(int,liste_entree))
10 print('Liste de nombres : ', liste_nombre)
11 print('Somme de la liste : ', sum(liste_nombre))
```

```
Entrez le nom, l'âge et le score de l'étudiant, séparés par des espaces : Jean 21 95
Nom : Jean
Âge : 21
Score : 95

Entrez une liste de nombres séparés par des espaces : 10 25 34 67 32
Liste obtenue par input : ['10', '25', '34', '67', '32']
Liste de nombres : [10, 25, 34, 67, 32]
Somme de la liste : 168
```

SCRIPT 19
Plusieurs valeurs sur plusieurs lignes

```
entrees = []
print("Entrez le nom des étudiants : ")

while True:
    etudiant = input()
    if etudiant:
        entrees.append(etudiant)
    else:
        break

print('Entrées reçues :')
print(entrees)
```

```
Enter Student Names:

John

Sophia

Philipe

Total input received :
['John', 'Sophia', 'Philipe']
```

## 3.2 Entrées graphiques

Avec le module `matplotlib`, il est possible de générer des graphiques. Il peut être intéressant alors de pouvoir sélectionner des points sur un plan pour ensuite utiliser ces points. La fonction utilisée pour déterminer des points est `ginput(n)`, où `n` est le nombre de cliques à faire. Puisque cette fonction fait partie du module `matplotlib.pylpot`, il faudra mettre un préfixe approprié pour l'appeler. Tel que mentionné dans la section sur le module, le nom du module est remplacé par un alias, dans ce cas-ci `plt`. Voici un exemple de code pour faire des entrées graphiques.

SCRIPT 20
Exemple de logiciel pour des entrées de données graphiques

```
# Importation des modules
import numpy as np
import matplotlib.pyplot as plt

# Création des objets pour générer une figure et un graphique
fig, ax = plt.subplots()
# Détermine le titre du graphique
ax.set_title("Exemple d'entrée graphique")
# Impose les limites des axes x et y
plt.xlim([0,1])
plt.ylim([0,1])

# Prends 3 cliques sur le graphique et transforme les données en matrice (array)
x = np.asarray(plt.ginput(3))

# Affichage des points
ax.plot(x[:,0],x[:,1],'r',lw=2)

# Affiche la figure
plt.show()
```

Il est important de noter que pour avoir une sortie sur une figure, il faut avoir le bon *backend*, dans ce cas-ci *Qt5*. Veuillez vous référez ici pour la procédure.

## 3.3 Sorties dans la console

Les sorties les plus communes sont à travers la console. Ces sorties permettent d'afficher des résultats, des messages et des informations sur les erreurs dans le code ou les entrées. Dans cette section, les affichages simples et formattés seront abordés et finalement d'autres sortes de sorties consoles pertinentes.

### 3.3.1 Affichage simple : `print()`

L'affichage le plus simple est d'inscrire la fonction `print()` avec une ou des chaînes de caractères comme argument d'entrée. Il est à noter que la fonction `print()` peut prendre autre chose qu'une chaîne de caractères en entrée seulement si il s'agit du seul argument. Par exemple, l'appel `print(34)`

sortira le nombre 34 ; par contre, l'appel `print('Nombre: ' + 34)` retournera une erreur. Il faut donc absolument transformer en chaînes de caractères les entrées que l'on souhaite concaténer.

### 3.3.2 Affichage formatté

L'affichage formatté est ce qui permet de sortir dans la console des informations qui ne sont pas nécessairement constantes ou que l'on souhaite formatter d'une manière bien précise en limitant la quantité de code nécessaire. Dans la sous-section sur l'affichage simple, on a pu voir qu'il est possible de concaténer directement des chaînes de caractères à l'aide de `+` dans l'appel de fonction `print()`, mais cela peut rapidement devenir éreintant et peu lisible. Il devient alors pertinent d'utiliser une méthode qui permet au code d'être plus lisible et plus rapide à écrire. Seulement deux manières seront explicitées, mais il en existe beaucoup plus.

Ainsi, il existe quelques méthodes d'affichages formatté. Il y a d'abord la méthode utilisant le symbole `%`. Voici un appel classique utilisant cette méthode : `print('Bienvenue %s, vous avez %d ans et mesurez %.2f mètre' % (nom,age,taille))`. Décortiquons cet appel. L'argument `%s` indique qu'à cet endroit, il y aura une chaîne de caratères qui sera introduite. L'argument `%d` indique qu'un nombre entier sera introduit. L'argument `%.2f` indique qu'il y aura un nombre flottant sera introduit, avec une précision de deux chiffres après la virgule. Le symbole `%` suivant la chaîne de caractères indique que l'information qui suit est ce qui doit apparaître, dans l'ordre d'écriture, pendant l'affichage. Le tuple `(nom,age,taille)` contient les valeurs que l'on souhaite ajouté. Bien qu'il soit possible de coder en dure les valeurs dans le tuple, il est beaucoup plus courant d'inscrire des noms de variables dans le tuple. Le tuple lui-même peut être généré au préalable pour ensuite faire son appel. Par exemple, on peut stocker les informations `nom,age,taille` dans le tuple `infos` et ainsi faire l'appel de cette manière : `print('Bienvenue %s, vous avez %d ans et mesurez %.2f mètre' % infos)`

La seconde manière utilise la méthode `format()`. Le principe de base reste le même. Il y a en entrée une chaîne de caractères avec les endroits réservées pour insérer les valeurs, représenté dans ce cas-ci par des accolades `{}`. À la suite de la chaîne de caractères, on indique les valeurs que l'on souhaite insérer. Voici un exemple : `print('Bienvenue {} ! Vous avez {} ans.'.format(nom,age))`. Les valeurs seront déterminé dans l'ordre, à moins d'indication contraire. Le formattage des valeurs se fait à l'intérieur des accolades. Cette méthode est très extensive et les subtilités ne seront pas discutées. Cependant, le script 21 possède plusieurs exemples qui peuvent être examinés pour de plus amples informations.

Le tableau 3.3.2 regroupe les opérateurs qui peuvent être utilisés dans les deux méthodes afin d'afficher ce qui est désiré.

| Opérateur | Description |
|---|---|
| `s` | L'affichage se fera telle une chaîne de caractères |
| `d` | L'affichage se fera tel un entier |
| `f` | L'affichage se fera tel un nombre flottant, avec le précision de base |
| `.3f` | L'affichage se fera tel un nombre flottant, avec une précision de 3 décimales. Le nombre peut être changé pour n'importe quelle précision souhaitée |

### 3.3.3 Les sauts de ligne, l'alignement, et affichage de tableaux

Les sauts de lignes lors de l'affichage se fait grâce à la chaîne de caractères
`n`. Cette chaîne se place à l'endroit exact où le saut de ligne est souhaité, à l'intérieur de la chaîne de caractères. Il n'est pas nécessaire d'isoler la chaîne avec des espaces blancs.

Script 21

```
# Méthode format()
nom = 'Tristan'
age = 23
taille = 1.72

# format sans index ni formattage
print('Bienvenue {} ! Vous avez {} ans et mesurez {} mètre.'.format(nom,age,taille))

# format avec index; 0 représente le premier élément inscrit dans les argument
    d'entrées de format()
# 1 représente le deuxième élément, etc
print('Bienvenue {0} ! Vous avez {2} ans et mesurez {1} mètre.'.format(nom,taille,age))

# format avec index et format
# À NOTER : Le modulo (%) est remplacé par l'index et le deux-point
print('Bienvenue {0:s} ! Vous avez {2:d} ans et mesurez {1:.3f}
    mètre.'.format(nom,taille,age))

# format avec nom de variables; les variable doivent être déclarées dans l'argument
    d'entrée de format()
print('Bienvenue {nom:s} ! Vous avez {age:d} ans et mesurez {taille:.3f}
    mètre.'.format(nom='Joseph',taille=1.45,age=21))
```

```
Bienvenue Tristan ! Vous avez 23 ans et mesurez 1.72 mètre.
Bienvenue Tristan ! Vous avez 23 ans et mesurez 1.72 mètre.
Bienvenue Tristan ! Vous avez 23 ans et mesurez 1.720 mètre.
Bienvenue Joseph ! Vous avez 21 ans et mesurez 1.450 mètre.
```

L'alignement est un peu plus compliqué. Commençons par la tabulation. La tabulation consiste à placer le curseur à des points discrets sur la ligne, tous ayant la même distance entre eux. Si le curseur dépasse un point, la tabulation se fera au prochain point. Par défaut, la tabulation maximale est de 8 caractères. L'application d'une tabulation se fait par l'insertion de la chaîne de caractères `t` à l'endroit souhaitée. Il n'est pas nécessaire d'isoler non plus.

L'alignement direct, pour avoir du texte aligné à droite ou au centre par exemple, se fait plus aisément avec la méthode `.format()`. En effet, il suffit d'insérer dans les accolades les symboles appropriés pour les différents alignements. Le symbole pour le texte centré est `^`, pour l'alignement à gauche `<` et pour l'alignement à droite `>`. Les espaces blancs peuvent être remplacer par le caractère au choix du codeur en plaçant ce caractère juste avant le symbole d'alignement. Finalement, pour que l'alignement corresponde à ce qui est voulu, il faut indiquer à l'interpréteur l'alignement se fait sur combien de caractères. En effet, la console n'a pas de limite intrinsèque et donc l'alignement doit se faire dans un nombre de carcatères. Ce nombre est indiqué juste après le symbole d'alignement.

L'alignement peut également se faire une méthode d'un objet. La méthode `center()` permet de centrer un texte selon la largeur en nombre de caractères entrée en argument. La méthode `ljust()` permet l'alignement justifié à gauche, avec encore en entrée la largeur en nombre de caractères. Finalement, la méthode `rjust()` permet l'alignement justifié à droite, selon le même principe que les deux précédentes méthodes.

L'affichage de tableau se fait en manipulant l'ensemble des méthodes de mise en page et de formattage énoncées. Il existe bien sûr des modules permettant de faire plus efficacement l'affichage, mais ils ne sont pas natifs à Python et doivent donc être indépendamment installés.

### 3.3.4 Comment lire les erreurs

Le codeur sera régulièrement confronté à des erreurs. Pour pouvoir les corriger, il est important de savoir lire les messages d'erreurs qui s'affichent dans la console. Typiquement, la console fournira le lieu où une erreur s'est trouvée, donc le fichier, la ligne et la commande qui a généré l'erreur. Ensuite, le type d'erreur est affichée, avec une explication sommaire de l'erreur. Étant donné que la communauté de programmation est largement anglophone, les messages sont en anglais.

```
Traceback (most recent call last):

  File "<ipython-input-7-132112355731>", line 1, in <module>
    random.rand(2)

AttributeError: module 'random' has no attribute 'rand'
```

FIGURE 3.1 – Exemple typique de message d'erreur

Dans la figure 3.1, il est possible de voir les informations d'une erreur. Le fichier, *file*, est un nom étrange car il s'agit d'une erreur suite à une commande directe dans la console. Si cela avait été un script, le chemin du fichier serait affiché au complet. Ensuite, la ligne, *line*, où se trouve l'erreur est affichée, dans ce cas-ci, l'erreur se trouve à la ligne 1. En jaune, *random.rand(2)*, se trouve la commande qui a généré l'erreur. En rouge, *AttributeError*, est le type d'erreur, avec ensuite l'explication sommaire.

En portant attention aux messages d'erreurs, il est considérablement plus facile de corriger les erreurs et de progresser rapidement dans l'élaboration de script.

SCRIPT 22
Démonstration de mise en page d'un tableau

```
# Données du tableau
donnees = [['Nom','Age','Latéralité','Grandeur'],
           ['Jean',38,'G',1.56],
           ['Robert',29,'D',1.84],
           ['Emma',37,'D',1.67],
           ['Simon',34,'G',1.54],
           ['Gary',22,'D',1.36]]

# Création du séparateur
tiret = '-' * 38

# Affichage de l'entête avec alignement centré
print("Tableau d'informations".center(38))

for i in range(len(donnees)):
    if i == 0:
      print(tiret)
      print('{:<10s}{:>4s}{:>12s}{:>12s}'.format(donnees[i][0],donnees[i][1],
                                              donnees[i][2],donnees[i][3]))
      print(tiret)
    else:
      print('{:-<10s}{:>4d}{:^12s}{:>12.2f}'.format(donnees[i][0],donnees[i][1],
                                                donnees[i][2],donnees[i][3]))
```

```
       Tableau d'informations
--------------------------------------
Nom       Age  Latéralité   Grandeur
--------------------------------------
Jean------ 38    G            1.56
Robert---- 29    D            1.84
Emma------ 37    D            1.67
Simon----- 34    G            1.54
Gary------ 22    D            1.36
```

## 3.4 Lecture et écriture d'un fichier

Afin de pouvoir faire du traitement de données et publier les résultats, il est important de savoir charger des fichiers et de savoir écrire dans des fichiers. À ce titre, cette section montrera des méthodes de chargement pour différents types de fichiers et avec différents outils.

### 3.4.1 Charger un fichier externe

Commençons avec les méthodes natives à Python pour charger des fichiers. De manière générale, la fonction `open('nom_de_fichier','mode')` sera utilisée pour ouvrir le fichier désiré dans un objet qui pourra ensuite être traité. Le nom de fichier et le mode doivent être considérées comme des chaînes de caractères. Une fois le traitement terminé, le fichier devra être fermé avec la méthode `close()`. Si cela n'est pas fait, il est possible que des problèmes de modification de fichier, d'ouverture ou de déplacement se présentent.

#### Fichier *.txt*

Un type de fichier relativement simple à charger est le fichier texte, avec l'extension `.txt`. La méthode de chargement pour la lecture se résume à ouvrir le fichier dans un objet, lire le fichier avec les fonctions appropriées et fermer le fichier. Le script 23 est un exemple simple. L'ouverture se fait par la fonction `open()` avec en argument le nom du fichier souhaité et le mode d'ouverture, ici en lecture. La lecture se fait ici par la méthode `readlines()`. Cette méthode lit l'entièreté du document et place chaque ligne dans une chaîne de caractères. Le fichier est ensuite fermé et le traitement de données peut se faire par la suite. Le traitement dépendra du format du fichier chargé, ce qui doit être connu à l'avance.

Script 23
Exemple de chargement pour lecture d'un fichier texte

```
#%% Lecture Fichier .txt
# Ouverture d'un fichier .txt en mode lecture, fichiertxt étant l'objet
fichiertxt = open('mri.txt','r')

# Lecture du fichier avec la méthode readlines()
contenu = fichiertxt.readlines()

# Fermeture du fichier
fichiertxt.close()

# Exemple de traitement de données
premiereLigne = contenu[0].split()
data = []

for i in range(1,len(contenu)):
    data.append(contenu[i].split())
```

Il existe principalement trois méthodes de lecture : `read()`, `readline()` et `readlines()`. La méthode `read()` permet de lire les bits d'un fichier. Il est possible de mettre en argument combien de bits on souhaite lire et la valeur par défaut est `-1`, ce qui équivaut à l'entièreté du document. Il est

à noter que chaque utilisation de la méthode `read()` place ce qui est obtenue dans une seule chaîne de caractères. La méthode `readline()` retourne dans une chaîne de caractères la ligne où se trouve le curseur au moment de l'appel. Il est possible de mettre en argument le nombre de bits que l'on souhaite obtenir de la ligne lue, avec la valeur par défaut de `-1` pour tout le document. La méthode `readlines()` retourne chaque ligne du fichier dans une chaîne de caractères et toutes les chaînes sont rassemblées dans une liste. Il est possible de mettre en argument le nombre maximum de bits devant être lus. Une fois le nombre de bits excédé, aucune autre ligne ne sera retournée.

### Fichier *.csv*

Un type de fichier relativement commun pour faire du traitement de données est le fichier de type CSV. L'acronyme provient de l'anglais *comma-separated values* et ce type est souvent généré par des tableurs. Bien que le nom implique que les valeurs sont séparées par des virgules, cela n'est pas une obligation absolue et différents séparateurs peuvent être utilisés, tant et aussi longtemps que le choix est constant à travers l'entièreté du document.

Ce type de fichier peut être charger et traité de la même manière qu'un fichier texte en utilisant les fonctions et méthodes internes de Python. Cependant, les séparateurs seront considérés comme faisant partie des chaînes de caractères et devront être pris en considération lors du traitement. Il existe un module spécifique pour les fichiers CSV, portant le nom `csv`, avec ses propres fonctions et méthodes pour la lecture et l'écriture. Cela pourra devenir pertinent lorsque plusieurs dialectes sont utilisées.

SCRIPT 24
Exemple de chargement d'un fichier CSV, avec une déclaration `with` et sans le module `csv`

```
#%% Lecture Fichier .csv avec déclaration with
with open('cities.csv','r') as fichiercsv:
    contenu = fichiercsv.readlines() # Lecture du fichier
    fichiercsv.close()               # Fermeture du fichier

# Exemple de traitement de données
premiereLigne = contenu[0].split(',') # Séparation des données aux virgules
data = []

for i in range(1,len(contenu)):
    data.append(contenu[i].split(',')) # Séparation des données aux virgules
```

### Écriture

Il est intéressant de pouvoir écrire dans un fichier pour faire de la publication ou de la distribution. Il y a trois modes d'ouverture de fichier permettant l'écriture : `w`, `a` et `x`. Le mode `w` place le curseur au début du fichier et écrase le contenu déjà présent. Si le fichier n'existe pas, il sera automatiquement créé. Le mode `a` place le curseur à la fin du document et continue l'écriture du fichier. Si le fichier n'existe pas, il sera automatique créé. Le mode `x` crée un fichier et permet l'écriture dans ce dernier. Si le fichier existe déjà, la console retournera une erreur indiquant que le fichier existe déjà. L'écriture elle-même est fait avec la méthode `write()`, prenant en argument la chaîne de caractères à insérer dans le document.

Le formattage de l'écriture utilise les mêmes normes que pour les sorties consoles.

SCRIPT 25
Exemple d'écriture dans un fichier texte

```
#%% Écriture Fichier .txt
# Création de données pour enregistrement
x = [1,2,3,4,5,6,7,8,9]
y = [4,2,6,4,9,1,6,3,7]

# Ouverture d'un fichier .txt en mode écriture
fichiertxt = open('exemple.txt','w')

# Écriture dans le fichier
fichiertxt.write('x \t y\n') # Écriture de l'entête

for i in range(0,len(x)):
    fichiertxt.write('%d \t %d\n' % (x[i],y[i])) # Écriture des données

# Fermeture du fichier
fichiertxt.close()
```

### 3.4.2 Charger un fichier texte avec `numpy`

Avec le module `numpy`, il est possible de charger des fichiers de manière très efficace. En effet, il n'est plus nécessaire d'ouvrir et fermer les fichiers, il suffit d'appeler la fonction appropriée. La lecture et l'écriture possédent leur propre fonction.

#### Lecture d'un fichier texte : `np.loadtxt()`

Tout d'abord, il est possible de charger un fichier avec la fonction `np.loadtxt('nom_de_fichier', dtype = float)`. Le nom du fichier doit être inscrit comme une chaîne de caractères et l'extension (`.txt`,`.csv`, etc) doit également y être présente. L'argument `dtype` indique de quelle manière les valeurs dans le fichier doivent être traitées, avec le type par défaut étant `float`. Un autre argument pouvant être pratique est `delimiter`. La valeur par défaut est l'espace blanc. Cependant, certains fichiers utilisent des virgules ou des points-virgules, il est alors intéressant de pouvoir spécifier à la fonction comment séparé les valeurs. La fonction retourne un vecteur multidimensionnel contenant les valeurs.

#### Écriture d'un fichier texte : `np.savetxt()`

Ensuite, il est possible d'enregistrer les résultats obtenus dans un fichier texte. La fonction utilisée est `np.savetxt('nom_de_fichier', donnees, fmt='%.6e', delimiter=' ')`. Celle-ci permet de gagner beaucoup de temps. En effet, contrairement aux méthodes plus traditionnelles natives à Python, il n'est plus nécessairement de faire le formattage manuellement. Au lieu de prendre plusieurs lignes de code pour écrire dans un fichier, une seule commande est nécessaire. Le nom de fichier et son extension est d'abord inscrit en chaîne de caractères. Ensuite, le vecteur ou la matrice à enregistrer est inscrite, représenté par `donnees`. Le format d'écriture, `fmt`, peut être déterminé à la préférence du codeur, mais c'est un argument optionnel. Le format est inscrit sous forme d'une chaîne de caractères et utilise les mêmes normes que pour les sorties consoles. Ce qui sépare les colonnes, `delimiter`, peut également être déterminé au choix du codeur, comme étant une chaîne de caractères. Cet argument est également optionnel.

| | |
|---|---|
| `num` | Cet argument peut être un nombre entier ou une chaîne de caractères. Si c'est une chaîne de caractère, cela devient également le titre de la fenêtre. |
| `figsize` | Cet argument doit prendre 2 valeurs `float`, sous forme d'un tuple, d'une liste ou d'un vecteur. Le premier nombre représente la largeur, le deuxième la hauteur. Les unités sont le pouce. Valeurs par défaut : (6.4 , 4.8) |
| `dpi` | Cet argument permet de changer la résolution de la figure, en points par pouce. Valeur par défaut : 100.0 |
| `facecolor` | Cet argument permet de modifier la couleur de l'arrière-plan de la figure |
| `edgecolor` | Cet argument permet de modifier la couleur de la bordure de la figure |
| `frameon` | Cet argument prend une valeur booléenne. Si `True`, le cadre s'affiche. Si `False`, le cadre ne s'affiche pas, ce qui rend les arguments `facecolor` et `edgecolor` caducs. Valeur par défaut : `True` |
| `clear` | Cet argument prend une valeur booléenne. Si `True` et que la figure existe déjà, le contenu précédent est supprimé. Si `False` et que la figure existe déjà, le contenu précédent est préservé, mais peut tout de même être écrasé |
| `tight_layout` | Cet argument prend une valeur booléenne. Si `True`, les graphiques sont auto-ajustés pour être entièrement lisibles dans la figure. Valeur par défaut : `False` |
| `constrained_layout` | Cet argument prend une valeur booléenne. Fonctionnelement identique à `tight_layout` |

TABLE 3.1 – Liste non-exhaustive d'arguments possibles pour la fonction `plt.figure()`

## 3.5 Sorties graphiques

Il est important de pouvoir faire une visualisation direct des données. Pour ce faire, il existe une librairie pour Python qui s'appelle `matplotlib`. Cette dernière permet d'obtenir énormément de contrôle sur ce qui est présenté. Afin de simplifier la présentation de figures, seul `matplotlib.pyplot` sera présenté. De plus, il est d'usage de donner l'alias `plt` afin de raccourcir les préfixes subséquents. Dans cette section, l'affichage de différents types de graphiques sera abordé, pour ensuite voir comment gérer l'affichage de plusieurs graphiques à l'intérieur de la même figure et finalement voir comment gérer les attributs d'un graphique et d'une figure.

### 3.5.1 Affichage d'une courbe : `plt.plot()`

Un graphique qui est couramment rencontré est la courbe plane. Il existe 2 méthodes différentes pour créer des figures et des graphes à l'aide de `matplotlib.pyplot` : la méthode orientée objet et la méthode pyplot. Les deux méthodes seront présentées.

La méthode pyplot consiste à d'abord créer une figure avec la fonction `plt.figure()`. La fonction peut prendre plusieurs arguments, permettant d'affecter un identifiant, de modifier les dimensions de la figure, la résolution, la couleur de l'arrière-plan et de la bordure, ainsi que plusieurs autres.

Ensuite, il suffit de créer la courbe avec la fonction `plt.plot(x,y)`, avec `x` les valeurs de l'axe des abcisses et `y` les valeurs de l'axes des ordonnées. Il est possible de créer plusieurs courbes dans un seul appel de fonction ou dans plusieurs appels de fonctions. Chaque courbe créée peut avoir son propre formatage ou esthétique, qui est indiqué après les données à l'aide d'une chaîne de caractères. Les formatages possibles seront vu plus en profondeur dans cette sous-section. De plus, les courbes peuvent être individuellement identifiées en ajoutant le mot-clé `label` en argument et en entrant une chaîne de caractère. Cela permettra de générer une légende si nécessaire et de faire les bonnes associations.

Une fois la courbe créée, il est possible d'ajouter un titre, des étiquettes pour les axes, une légende et une grille. Le titre est ajouté avec la fonction `plt.title()` et prend en argument une chaîne de caractères. Les étiquettes d'axes sont ajoutées avec les fonctions `plt.xlabel()` et `plt.ylabel()` et les deux fonctions prennent en argument une chaîne de caractères. La création de la légende se fait par la fonction `plt.legend()`. Si les courbes ont leur identifiant créés lors du tracé, la fonction ne prend pas d'argument et tout se fera automatiquement. Si les courbes n'ont pas été identifiées, il est possible d'entrer en argument les identifiants dans une liste ou un tuple, dans l'ordre que les courbes ont été tracées.

Voici un exemple utilisant la méthode pyplot.

Script 26
Exemple de courbe avec la méthode pyplot

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Valeurs à utiliser
5  x = np.linspace(0,2)
6  y1 = x**1
7  y2 = x**2
8  y3 = x**3
9
10 # Création de la figure
11 fig = plt.figure(num='Figure de démonstration',figsize=(7,6))
12
13 # Traçage des courbes, avec leur identifiant
14 plt.plot(x,y1,label='Courbe linéaire')
15 plt.plot(x,y2,label='Courbe quadratique')
16 plt.plot(x,y3,label='Courbe cubique')
17
18 # Inscription du titre, des axes et de la légende
19 plt.title('Graphique de démonstration')
20 plt.xlabel('Axe x')
21 plt.ylabel('Axe y')
22 plt.legend()
```

La méthode orientée objet fonctionne un peu sur le même principe que la méthode pyplot. En effet, la première étape est de créer la figure. Cependant, dans la même étape, l'espace de données pour tracer les courbes est créé en même temps, en associant ces deux objets à des variables. La première commande est donc `fig, ax = plt.subplots()`, où `fig` est la variable référant à la figure et `ax` la variable référant à l'espace de données. La fonction `plt.subplots()` permet également de

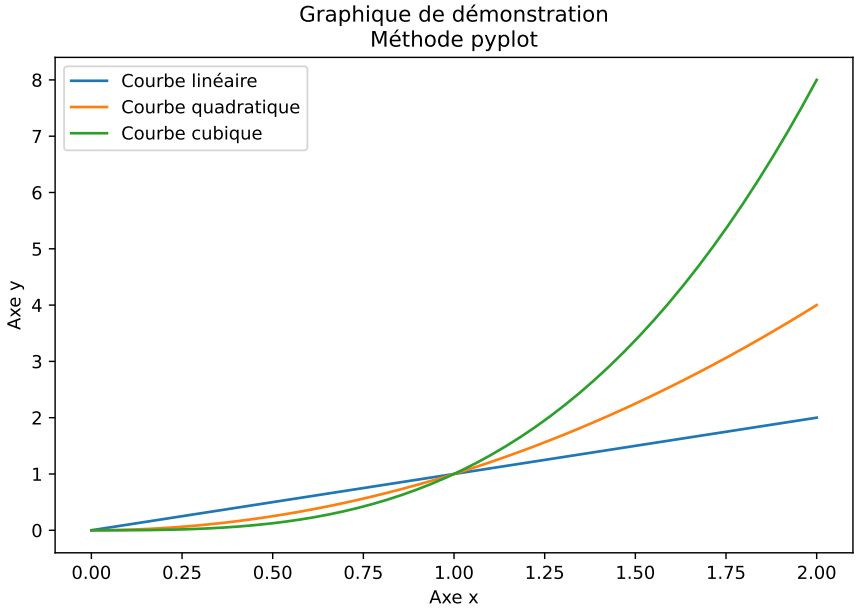


FIGURE 3.2 – La figure résultante du code d'exemple avec la méthode pyplot

produire des figures comportant plusieurs graphiques séparés, ce qui sera vu dans la sous-section sur l'affichage multiple.

Une fois les objets créés, les courbes peuvent être tracé de la même façon que la méthode pyplot, mais en remplaçant le préfixe `plt` par le nom de la variable approprié, dans ce cas-ci `ax`. Ainsi, la commande devient `ax.plot(x,y)`. Le formatage ainsi que les identifiants des courbes sont inscrits de la même manière que la méthode pyplot. Il en va de même pour la légende, qui devient `ax.legend()`.

L'ajout d'un titre et des identifiants des axes ne se fait pas de la même manière. À la différence de la méthode pyplot, la méthode orientée objet contient déjà un espace pour ces informations, il suffit donc de les remplir, à l'aide des fonctions `ax.set_title()`, `ax.set_xlabel()` et `ax.set_ylabel()`. Les arguments d'entrées demeurent cependant des chaînes de caractères.

Voici un exemple utilisant la méthode orientée objet.

SCRIPT 27
Exemple de courbe avec la méthode orientée objet

```
# Valeurs à utiliser
h = np.linspace(0,3)
g1 = (h**3)/8
g2 = (h**2)/4
g3 = 5*h-9

# Création de la figure et de l'espace de données
fig1, ax1 = plt.subplots()

# Traçage des courbes
ax1.plot(h,g1)
ax1.plot(h,g2)
ax1.plot(h,g3)

# Inscription du titre, des axes et de la légende
ax1.set_title('Graphique de démonstration\nMéthode orientée objet')
ax1.set_xlabel('Axe x')
ax1.set_ylabel('Axe y')
ax1.legend(['Courbe g1','Courbe g2','Courbe g3'])
```

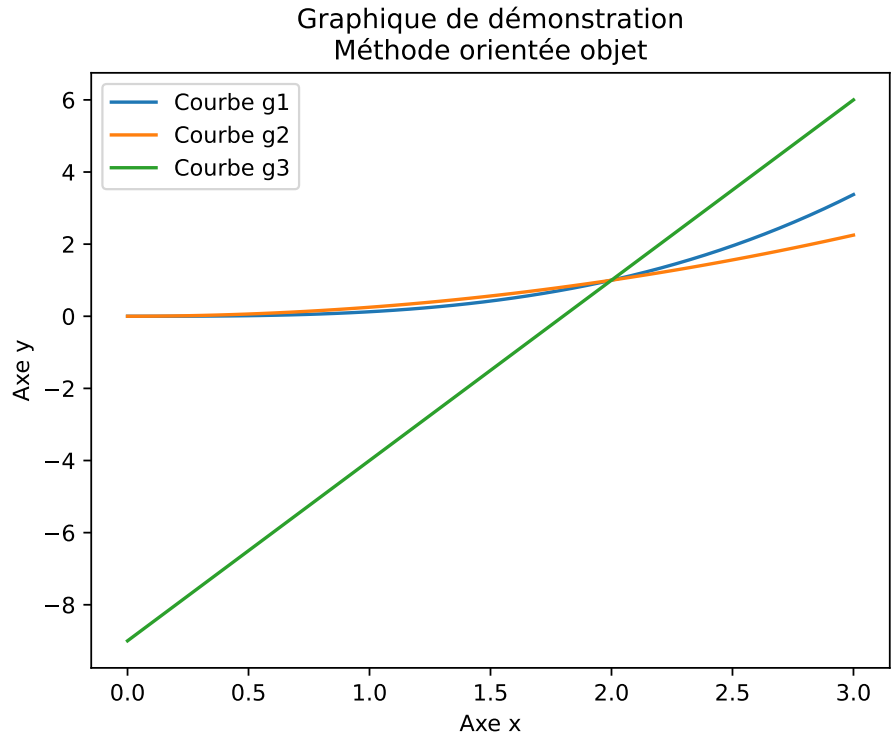


FIGURE 3.3 – La figure résultante du code d'exemple avec la méthode orientée objet

### 3.5.2 Affichage d'un graphique logarithmique

Un autre format de courbe qui est prévalent est le format logarithmique. Il est possible de former un seul des deux axes sur l'échelle logarithmique, les deux axes sur l'échelle logarithmique et de choisir la base pour chacun des axes.

Pour créer un graphique avec l'axe x possédant l'échelle logarithmique, la fonction à utiliser est `plt.semilogx(x,y)`. Pour obtenir un graphique dont l'échelle logarithmique est sur l'axe y, la fonction est `plt.semilogy(x,y)`. Pour créer un graphique dont les deux axes possèdent une échelle logarithmique, la fonction est `plt.loglog(x,y)`. Pour une utilisation avec la méthode orientée objet, il suffit de remplacer le préfixe `plt` par le nom de l'objet contenant l'espace de données.

Par défaut, l'échelle logarithmique est de base 10. Pour changer la base, il suffit de rajouter l'argument `base` et d'inscrire la base souhaitée. Cela fonctionnne pour toutes les fonctions d'affichage logarithmique. Par contre, dans le cas de `plt.loglog()`, cela change la base pour les 2 axes. Pour modifier l'échelle d'un seul axe, il est nécessaire de modifier la propriété d'une manière plus spécifique, avec soit `ax.set_xscale('log',base=2)` pour l'axe x en échelle logarithmique de base 2, soit `ax.set_yscale('log',base=3)` pour l'axe y. Il est à noter que le préfixe `ax` correspond à un objet. Ces méthodes pour changer les axes ne sont pas exclusives aux graphiques logarithmique et peuvent être utiliser pour d'autres courbes.

L'ajout de titre, d'identifiants pour les axes et d'une légende se fait de la même manière que pour une courbe standard, en faisant bien attention d'être constant dans la méthode utiliser, que ce soit pyplot ou orientée objet.

### 3.5.3 Affichage d'un histogramme : `plt.hist()`

Les histogrammes sont pratiques pour visualiser la distribution d'un ensemble de données. Afin de générer ce genre de graphique, la fonction à utiliser est `plt.hist(x,bins)`. L'argument d'entrée `x` est l'ensemble de données à traiter et l'argument d'entrée `bins`, le nombre de conteneurs équidistants.

Script 28
Exemple de création d'un histogramme

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # État aléatoire fixé pour reproduire les mêmes résultats
5  np.random.seed(1800000)
6
7  # Génération des données
8  mu, sigma = 100, 15
9  x = mu + sigma * np.random.randn(10000)
10
11 # Histogramme des données
12 plt.hist(x, 100)
13
14 # Paramètres du graphique
15 plt.xlabel('Axe x')
16 plt.ylabel('Axe y')
17 plt.title("Exemple d'histogramme")
```

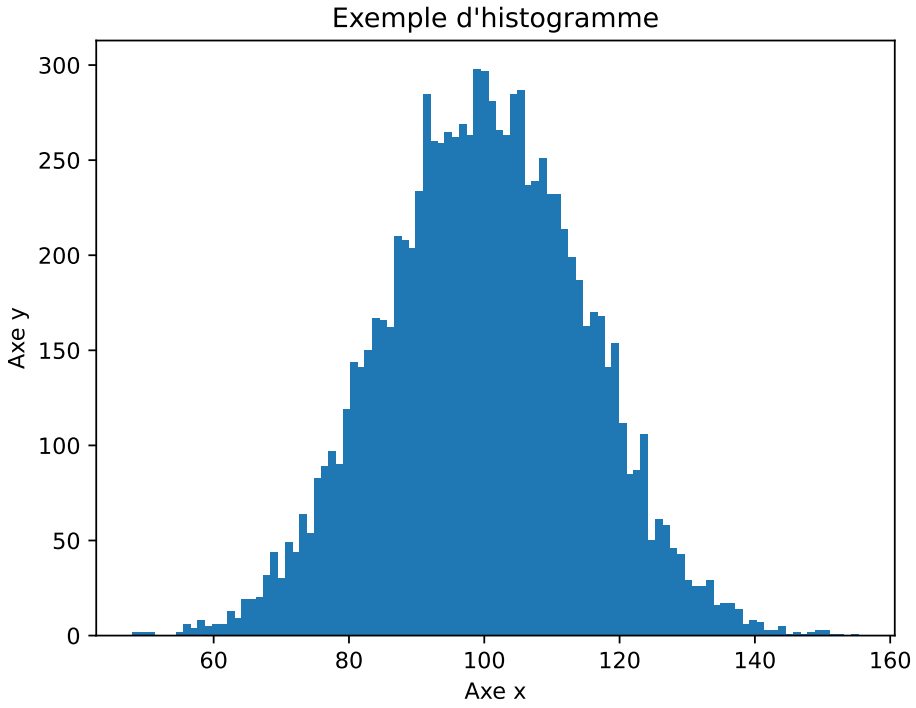


FIGURE 3.4 – L'histogramme résultant du code d'exemple

### 3.5.4 Affichage d'un nuage de point : `plt.scatter()`

Il arrive parfois qu'il y a un ensemble de données dont la relation est inconnue. Il est alors pratique de pouvoir visualiser cet ensemble. Pour ce faire, il est possible d'utiliser un graphique de nuage de points. La fonction permettant de générer ce type de graphique est `plt.scatter(x,y,s)`. Les arguments d'entrées `x` et `y` représentent les valeurs en x et en y respectivement, alors que l'argument d'entrée `s` est la grosseur des points.

SCRIPT 29
Exemple d'un nuage de points

```
import matplotlib.pyplot as plt
import numpy as np

# État aléatoire fixé pour reproduire les mêmes résultats
np.random.seed(1800000)

# Génération des données
x = np.arange(50)
y = x+10*np.random.randn(50)

# Création d'une figure et d'un espace de données
fig, ax = plt.subplots()

# Le nuage de points
ax.scatter(x,y,s=4)

# Paramètres du graphique
ax.set_xlabel('Axe x')
ax.set_ylabel('Axe y')
ax.set_title("Exemple d'un nuage de points")
```

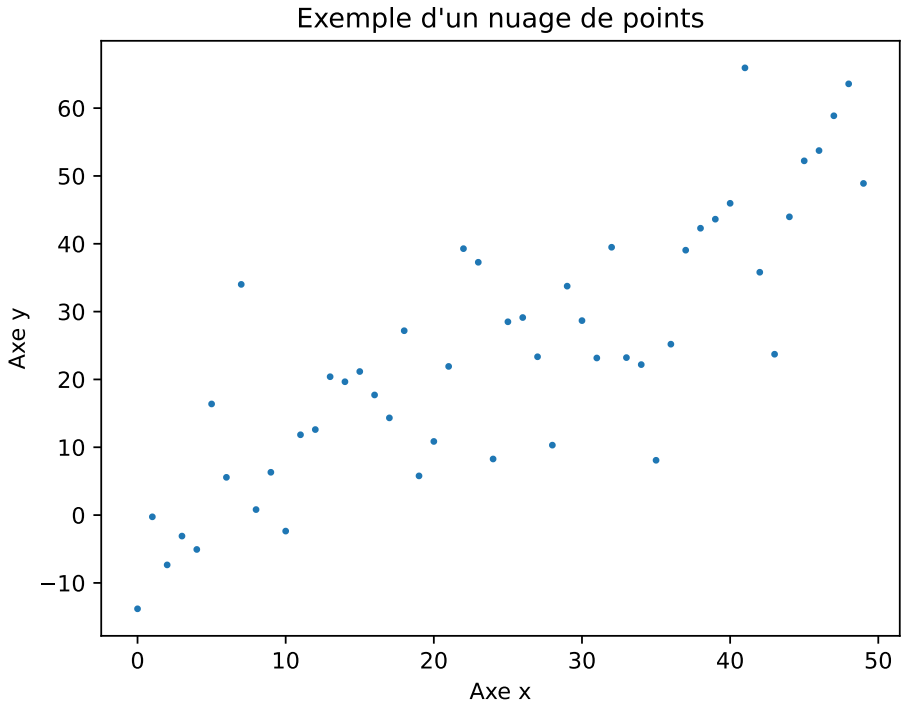


FIGURE 3.5 – La figure résultante de l'exemple d'un nuage de points

### 3.5.5 Affichage d'un graphique 3D

Avec Python et le module `matplotlib`, il est également possible de générer des figures 3D. Pour créer l'espace tridimensionnel, il faut cependant faire appel spécifiquement à un outil de `matplotlib`, `Axes3D`. La commande permettant son utilisation est `from mpl_toolkits.mplot3d import Axes3D`. Ensuite, il est possible de faire appel à l'outil dans le script.

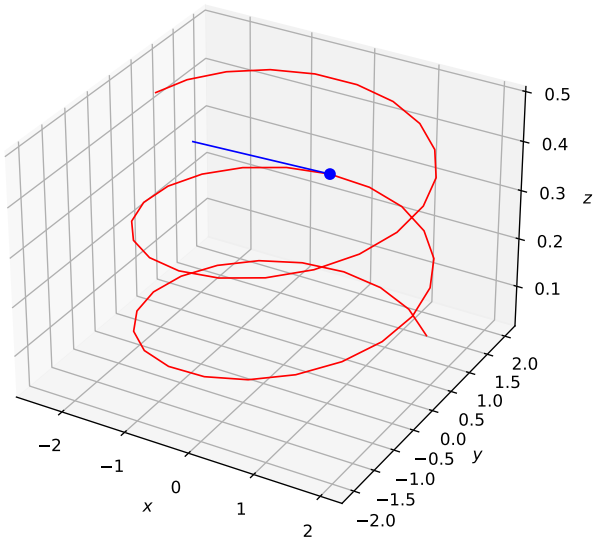


FIGURE 3.6 – Exemple de figure 3D

https ://matplotlib.org/stable/tutorials/toolkits/mplot3d.html

SCRIPT 30
Exemple de génération d'une figure 3D avec une hélice

```
#Programme pour creer et tracer une helice de rayon r et de pas S selon l'axe z,
#avec les equations parametriques x = r*cos(t), y = r*sin(t), z = s*t

#Importation des modules
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as nl
from mpl_toolkits.mplot3d import Axes3D

#Déclaration des parametres de l'helice
r, s = 2, 0.1/np.pi

#Générer le vecteur t
tmin, tmax = np.pi/4, 5*np.pi
npt=50 # Nombre des points
t=np.linspace(tmin, tmax, npt)

#Coordonnées de l'helice
x=r*np.cos(t)
y=r*np.sin(t)
z=s*t
p=np.array([x, y, z])

#transposée de la matrice p
p=p.T

#figure de l'helice en 3D
fig = plt.figure()
ax=Axes3D(fig)  #Génération de l'espace 3D dans la figure
ax.plot(p[:,0], p[:,1], p[:,2], linewidth=1.0, color='r')
ax.set_title('$ Helice $') #ou bien plt.title(' Helice ')
ax.set_xlabel('$x$')       #ou bien plt.xlabel('$x$')
ax.set_ylabel('$y$')       #ou bien plt.ylabel('$y$')
ax.set_zlabel('$z$')       #ou bien plt.zlabel('$z$')

#calculons et traçon la tagente au point 25
#en derivant le vecteur position rapport au parametre t: Vtan= [dxdt dydt dzdt]
dxdt = -r*np.sin(t)
dydt = r*np.cos(t)
dzdt = s
Vtan = np.array([dxdt, dydt,dzdt*np.ones(npt)])

#transposée de la matrice Vtan
Vtan=Vtan.T

#coordonnées de la tangente la position au point 25
Vtan_25 = Vtan[24]
p_25 = p[24]

#Graphe de la tangente
V1=p_25
V2=V1+Vtan_25
V=np.array([V1, V2] )

ax.plot(p_25[0],p_25[1],p_25[2],'o-', linewidth=15.0, color='b')
ax.plot(V[:,0], V[:,1], V[:,2], linewidth=1.0, color='b')
```

### 3.5.6 Affichage de plusieurs graphiques sur une même figure

Il est intéressant parfois d'obtenir plusieurs graphiques dans une même figure. Cela peut être pour des fins de présentation, de comparaison, de lisibilité, ou pour toutes sortes d'autres raisons. La manière d'arriver au résultat dépend de la méthode utiliser. Avec la méthode pyplot, la fonction utilisée sera `plt.subplot()`. Avec la méthode orientée objet, la fonction utilisée sera `plt.subplots()`. Il faut bien noter la différence entre les deux, l'un ayant un *s* et l'autre non. Les deux méthodes seront démontrées.

Avec la méthode pyplot, il est possible de concatener la forme et la position souhaitée. Par exemple, en écrivant `plt.subplot(231)`, on souhaite accéder au premier espace tel que si la figure était séparée en 2 lignes et 3 colonnes. En utilisant cette méthode, il n'est pas obligatoire de demeurer consistant dans l'écriture de la forme. Cependant, il est possible que les graphiques précédents soient écrasés par les graphiques subséquents si les espaces ne sont pas bien gérés. Il n'est pas obligatoire de concatener les valeurs, elles peuvent être séparées par des virgules. Le script 32 démontre la méthode.

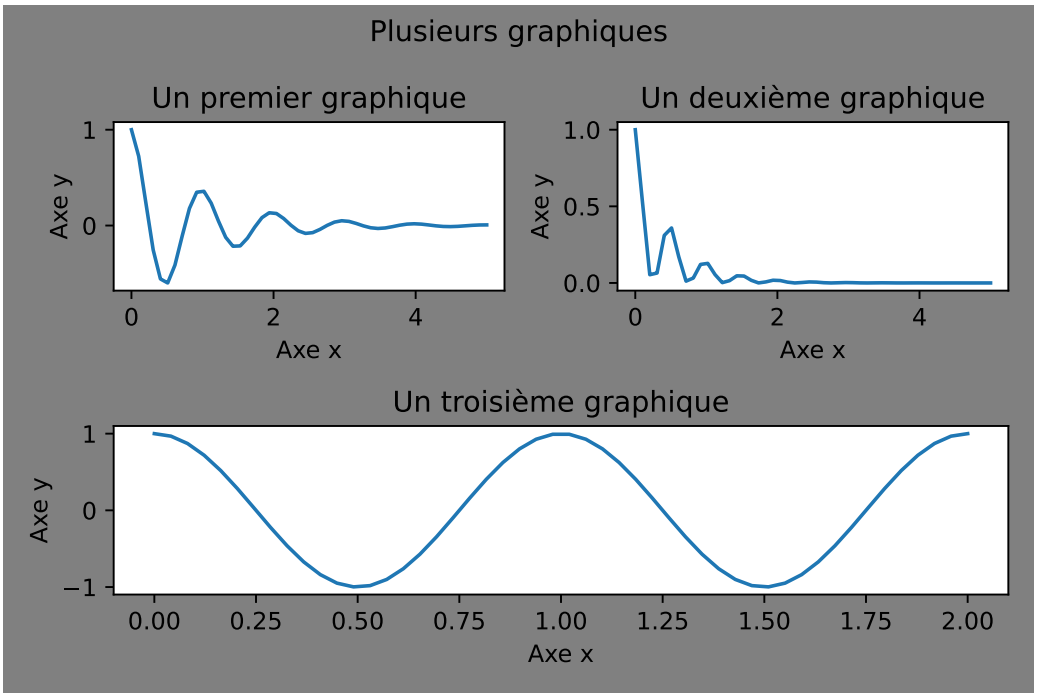


FIGURE 3.7 – Affichage multiple avec la méthode pyplot

Avec la méthode orientée objet, il suffit d'indiquer le format souhaité lors de la création de la figure et des espaces de données, par exemple en écrivant `plt.subplots(2,3)`. L'accès aux espaces de données se fait en indexant de la même manière qu'une matrice la variable. Le script **??** démontre un exemple.

SCRIPT 31
Exemple d'affichage multiple avec la méthode pyplot

```
import matplotlib.pyplot as plt
import numpy as np

# Données à utiliser
x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

# Création de la figure et du titre de la figure complète
plt.figure('Figure à affichage
    multiple',tight_layout=True,facecolor='grey',edgecolor='black')
plt.suptitle('Plusieurs graphiques')


# Traçage du premier graphique, en première position selon un arrangement 2x2
plt.subplot(221) # Exemple de format et position concatené
plt.plot(x1,y1)
plt.xlabel('Axe x')
plt.ylabel('Axe y')
plt.title('Un premier graphique')

# Traçage du deuxième graphique, en 2e position selon un arrangement 2x2
plt.subplot(222)
plt.plot(x1,y1**2)
plt.xlabel('Axe x')
plt.ylabel('Axe y')
plt.title('Un deuxième graphique')

# Traçage du troisième graphique, en 2e position selon un arrangement 2x1
plt.subplot(2,1,2) # Exemple de format et position séparé par des virgules
plt.plot(x2,y2)
plt.xlabel('Axe x')
plt.ylabel('Axe y')
plt.title('Un troisième graphique')
```

SCRIPT 32
Exemple d'affichage multiple avec la méthode orienté objet

```
import matplotlib.pyplot as plt
import numpy as np

# Données à utiliser
x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

# Création de la figure et des espaces de données, arrangement 2x2
fig1, ax = plt.subplots(2,2)
fig1.suptitle('Plusieurs graphiques\nMéthode OO')

# Traçage des graphiques
ax[0,0].plot(x1,y1)      # Position en haut à gauche
ax[0,0].set_xlabel('Axe x')
ax[0,0].set_ylabel('Axe y')
ax[0,0].set_title('Premier graphique')

ax[0,1].plot(x1,y1**3)   # Position en haut (0) à droite (1)
ax[0,1].set_xlabel('Axe x')
ax[0,1].set_ylabel('Axe y')
ax[0,1].set_title('Deuxième graphique')

ax[1,0].plot(x2,y2)      # Position en bas (1) à gauche (0)
ax[1,0].set_xlabel('Axe x')
ax[1,0].set_ylabel('Axe y')
ax[1,0].set_title('Troisième graphique')

ax[1,1].plot(x2,np.cos(y2)) # Position en bas à droite
ax[1,1].set_xlabel('Axe x')
ax[1,1].set_ylabel('Axe y')
ax[1,1].set_title('Quatrième graphique')

# Commande assurant que les 4 graphiques rentrent dans la figure
plt.tight_layout()       # Important de placer APRÈS le traçage des graphiques
```

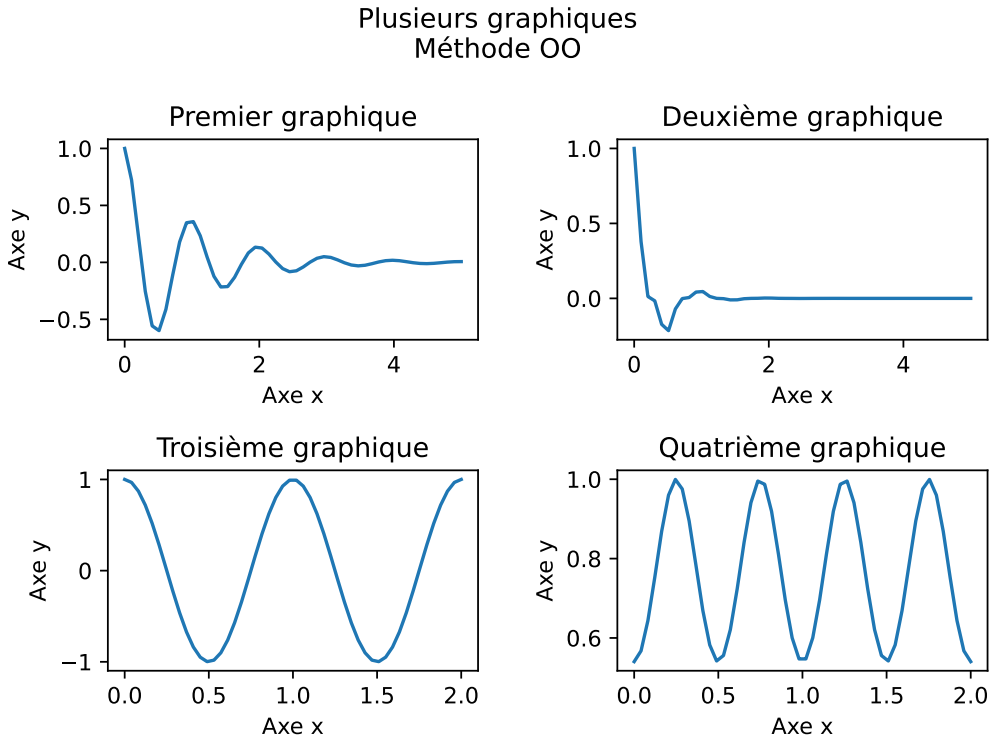


FIGURE 3.8 – Affichage multiple avec la méthode pyplot

### 3.5.7 La couleur, les marqueurs et la forme

Lors de la création d'un graphique, il est très souvent pertinent d'avoir le contrôle sur l'apparence. Pour une courbe, il est possible de déterminer la grosseur, la couleur et la forme du trait et la couleur, la forme et et la grosseur des marqueurs.

Lors du tracé, il est possible de changer ces paramètres. Le tableau 3.2 référence les mots-clés utilisés pour changer les paramètres. Le tableau 3.3 référence les couleurs usuelles. Il y a une liste plus complète ici :

| | |
|---|---|
| `ls` | Le style du tracé. Doit être écrit sous la forme d'une chaîne de caractères. Valeurs possibles : `'-','--','-.',':','None'` |
| `lw` | La largeur du tracé. Doit être un nombre réel. |
| `color` | La couleur du tracé. Doit être écrit sous la forme d'une chaîne de caractères. Il y a plusieurs ensembles de couleurs, notamment les couleurs nommées et les codes hexadécimals |
| `marker` | Le style du marqueur. Doit être écrit sous la forme d'une chaîne de caractères. Valeurs possibles : `'.','o','v','s','p','*','x'`, etc |
| `markersize` | La grosseur du marqueur. Doit être un nombre réel. |

TABLE 3.2 – Tableau des mots-clés pour changer les paramètres des graphiques

| | |
|---|---|
| `b` | Bleu |
| `g` | Vert |
| `r` | Rouge |
| `c` | Cyan |
| `m` | Magenta |
| `y` | Jaune |
| `k` | Noir |
| `w` | Blanc |

TABLE 3.3 – Tableau de quelques couleurs nommées