2024 - 2025

Programmation numérique, analyse de données

Pierre Sochala

École Spéciale des Travaux Publics, du Bâtiment et de l'Industrie

Table des matières

1 Introduction

La **programmation** ou codage désigne l'ensemble des activités qui permettent l'écriture des programmes informatiques. La programmation s'appuie sur des **algorithmes**, qui sont des suites finies d'instructions logiques, pour écrire des **programmes** destinés à être exécutés par un ordinateur. Plus précisément, un algorithme est un concept abstrait exprimé en langage générique tandis qu'un programme est l'implémentation de ce concept dans un langage informatique particulier. Cette distinction nous montre d'emblée que la programmation numérique est une activité pluridisciplinaire nécessitant a minima des compétences en informatique pour écrire des programmes dans un langage approprié et en mathématiques pour développer des algorithmes de calcul.

Un ordinateur est défini comme une machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques. L'information réellement traitée par un ordinateur est cependant très éloignée de celle fournie dans les programmes informatiques puisqu'un ordinateur ne traite que des suites de nombres binaires (0 ou 1) traités par groupes de 8 (les « octets »), 16, 32 ou 64. Pour qu'un programme informatique fonctionne, le **code source** doit être transformé en code lisible par l'ordinateur. Deux approches existent : la **compilation** qui consiste à traduire le code source en un unique code binaire exécutable, et l'**interprétation** qui consiste à lire le code source en temps réel et exécuter chaque instruction l'une après l'autre. Les langages compilés sont plus rapides que les langages interprétés et sont donc préférés pour développer les systèmes d'exploitation, les logiciels et les jeux vidéos. Cependant, ces langages demandent plus de lignes de code et ont une syntaxe plus difficile à apprendre que celle des langages interprétés. De plus, l'exécutable dépend de la plate-forme. A l'inverse, les langages interprétés sont plus lents que les langages compilés mais ils offrent un code plus léger et flexible qui fonctionne sur toutes les plateformes. A titre illustratif, le `Pascal`, le `C` et le `C++` sont des langages compilés tandis que le `Python` et le `Java` sont des langages interprétés.

Un langage de programmation est un ensemble de mots-clefs associé à un ensemble de règles précises indiquant comment on peut assembler ces mots pour former des instructions que le compilateur ou l'interpréteur peut traduire en langage machine (binaire). L'objectif de ce cours est de donner les bases de la programmation en `Python`. Créé en 1991 par G. Van Rossum, le langage `Python` présente de nombreux points forts comme la simplicité, la popularité, la polyvalence et l'évolutivité. `Python` possède en effet une syntaxe lisible et directe qui s'apprend rapidement et évite aux novices de s'attarder sur des complexités de langage. Étant massivement utilisé, `Python` est pris en charge par la plupart des OS, et on dénombre une vaste quantité de librairies et d'APIs (application programming interface ou « interface de programmation d'application »). Par ailleurs, `Python` peut être utilisé pour le développement de logiciels, le développement web, l'apprentissage automatique, etc. De plus, chaque mise à jour du langage `Python` ajoute de nouvelles caractéristiques très utiles lui permettant de rester aligné avec les pratiques modernes de développement (cf. les différentes versions existantes de `Python`).

Les développements en `Python` peuvent se faire en utilisant des **scripts** ou des **carnets Jupyter**. La première option pour écrire et partager du code est avec des scripts. Un script est un fichier texte brut qui se termine par l'extension .py en `Python`. Pour créer et éditer des scripts `Python`, on peut utiliser un éditeur de texte standard (e.g. `Notepad++`, `emacs`, `vim`) ou un environnement de développement intégré (e.g. `Spyder`, `PyCharm`, `IDLE`). Un tel environnement, appelé IDE (pour integrated development environment), comporte plusieurs composantes facilitant les dévéloppements comme un éditeur de texte et de code source, un compilateur/interpréteur, un créateur d'interface graphique, des outils de tests automatiques, un débogueur, etc. La seconde option pour développer en `Python` est avec Jupyter notebook. Les carnets Jupyter sont des cahiers électroniques qui peuvent rassembler du texte, des images, des formules mathématiques et du code informatique exécutable. Ils sont manipulables interactivement dans un navigateur web. Un fichier notebook se présente comme une succession de cellules de deux types : i) des cellules « Markdown » permettant d'écrire du texte, d'insérer des images, et ii) des cellules « Code » permettant d'écrire des lignes de code en `Python` puis de les exécuter. Un carnet offre la possibilité de produire des blocs de code dans des cases indépendantes qui peuvent être testées individuellement.

2 Variables

2.1 Définition

Une variable en programmation informatique porte un *nom* et occupe une *zone mémoire* où est stockée une *valeur*. En `Python`, la déclaration d'une variable et son initialisation sont simultanées comme l'illustre l'exemple ci-après. La seule instruction de la ligne 1 permet d'effectuer les trois étapes suivantes en même temps : i) `Python` attribue le type entier à la variable `x`, on parle de typage dynamique (par opposition au typage statique qui est la règle par exemple en `C++`), ii) `Python` alloue l'espace en mémoire nécessaire au stockage d'un entier et associe le nom `x`, iii) `Python` assigne la valeur 2 à la variable `x`.

```
1 >>> x = 2
2 >>> x
3 2
```

2.2 Types

Il existe de nombreux types de variables comme les nombres décimaux appelés `float`, les chaînes de caractères nommées `string` ou `str`, les booléens désignés `bool`, etc. Nous pouvons observer sur les trois exemples suivant que la fonction `type()` renvoie le type d'une variable. Notons au passage qu'il est nécessaire d'entourer une chaine de caractères de guillemets.

```
1 >>> x=2.
2 >>> print(type(x))
3 <class 'float'>

4 >>> file_name='toto.txt'
5 >>> print(type(file_name))
6 <class 'str'>

7 >>> Test = 1==2
8 >>> print(type(Test))
9 <class 'bool'>
```

En programmation, on est amené à changer les types des variables, il s'agit de la conversion de types (appelée *casting* en anglais). Dans la gestion des noms de fichiers par exemple, on souhaite parfois passer d'un type numérique à une chaîne de caractères et réciproquement. Les fonctions `int()`, `float()` et `str()` permettent ces conversions de types en `Python`.

```
1 >>> i=2
2 >>> str(i)
3 '2'

4 >>> i=2.
5 >>> str(i)
6 '2.0'

7 >>> i='123'
8 >>> int(i)
9 123

10 >>> i='1.23'
11 >>> float(i)
12 1.23
```

2.3 Opérateurs

Les quatre opérations arithmétiques se font de manière simple sur les variables ayant un type numérique (entier ou flottant). L'opérateur puissance est symbolisée par $**$. Il existe également les opérateurs combinés `+=`, `-+`, `*=`, `/=` et `**=` qui effectuent une opération et une affectation en une seule étape.

```
1 >>> x=2
2 >>> x+1    # Addition
3 >>> x-3    # Soustraction
4 >>> x*4    # Multiplication
5 >>> x/5    # Division
6 >>> x**6   # Mise a la puissance n=6
7 >>> x**=2 # Mise au carre et affectation
8 >>> x
9 4
```

Deux opérations sont possibles pour les chaines de caractères : l'addition concatène et la multiplication duplique.

```
1 >>> chaine = "Hello "
2 >>> chaine + "World"
3 'Hello World'

4 >>> 2*chaine
5 'Hello Hello '
```

Il existe aussi les opérateurs de comparaison qui sont utilisés pour comparer deux variables. Ces opérateurs renvoient un booléen qui est un type de variable à deux états : vrai ou faux. Ces opérateurs sont très utiles pour écrire des conditions dans des structures de contrôle de flux comme les boucles `while`.

```
1 >>> x == y  # Egalite
2 >>> x != y  # Difference
3 >>> x > y   # superieur a
4 >>> x < y   # inferieur a
5 >>> x >= y  # superieur ou egal a
6 >>> x <= y  # inferieur ou egal a
```

3 Listes

3.1 Définition

Une liste est une structure de données qui contient une série de valeurs. Les éléments d'une liste sont séparés par des virgules, encadrés par des crochets et peuvent être de types différents en `Python`.

```
1 >>> Animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> Tailles = [5, 2.5, 1.75, 0.15]
3 >>> Mixte = ["girafe", 5 , "souris", 0.15]
```

3.2 Indices et Tranches

Un élément d'une liste est repéré par son indice entre crochets et l'indiçage commence à 0. Il est aussi possible d'utiliser un indiçage négatif en commençant à compter à partir de la fin. L'avantage de l'indiçage négatif est d'avoir accés au dernier élément d'une liste avec l'indice -1 sans connaître la longueur de la liste. L'indice -2 est associé à l'avant-dernier élément, l'indice -3 à l'antépénultième, etc.

```
1 >>> Animaux[0] = 'girafe'
2 >>> Animaux[3] = 'souris'
3 >>> Animaux[-1] = 'souris'
4 >>> Animaux[-2] = 'singe'
```

Une partie d'une liste peut aussi être facilement sélectionnée en utilisant la syntaxe `[m:n+1]` pour extraire les éléments du émième au énième (élément `m` inclus et `n+1` **exclus**). La partie extraite s'appelle une tranche de la liste. On peut omettre l'un des deux indices dans la selection. La syntaxe `[:n+1]` permet d'extraire le début de la liste jusqu'au énième élément tandis que la la syntaxe `[m:]`

permet d'extraire la fin de liste depuis le émième élément. On peut aussi préciser le pas d'extraction en ajoutant un symbole deux-points supplémentaire puis le pas (de type entier).

```
1 >>> animaux[0:2]
2 ['girafe', 'tigre']
3 >>> animaux[2:3]
4 ['singe','souris']
5 >>> animaux[:]
6 ['girafe', 'tigre', 'singe', 'souris']
7 >>> animaux[1:-1]
8 ['tigre', 'singe']
9 >>> animaux[0:3:2]
10 ['girafe', 'singe']
```

Il est possible de définir des listes de listes. Un simple indiçage permet d'accéder à une sous-liste et un double indiçage renvoie un élément d'une sous-liste.

```
1 >>> enclos1 = ["girafe",2]
2 >>> enclos2 = ["tigre",3]
3 >>> enclos3 = ["singe",4]
4 >>> zoo = [enclos1, enclos2, enclos3 ]
5 >>> zoo
6 [['girafe', 2], ['tigre', 3], ['singe', 4]]

7 >>> zoo[1]
8 ['tigre', 3]
9 >>> zoo[1][0]
10 'tigre'
11 >>> zoo[1][1]
12 3
```

3.3 Opérations

Comme pour les chaines de caractères, les listes supportent l'opérateur + qui concatène et l'opérateur ∗ qui duplique.

```
1 >>> Animaux1 = ["girafe", "tigre"]
2 >>> Animaux2 = ["singe", "souris"]
3 >>> Animaux1 + Animaux2
4 ["girafe", "tigre", "singe", "souris"]
5 >>> Animaux1*2
6 ["girafe", "tigre", "girafe", "tigre"]
```

3.4 Fonctions

La fonction `type()` peut s'appliquer aux listes ainsi que la fonction `len()` qui renvoie la longueur d'une liste, c'est-à-dire le nombre d'éléments qu'elle contient.

```
1 >>> type(Animaux)
2 list
3 >>> len(Animaux)
4 4
```

Une liste d'entiers peut facilement être obtenue en combinant la fonction `range()` qui permet de générer une séquence d'entiers et la fonction `list()` qui convertit une séquence, collection ou itérateur en liste. Les fonctions `min()`, `max()` et `sum()` donnent le minimum, le maximum et la somme d'une liste passée en argument.

```
1 >>> Entiers = list(range(0,10))
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 >>> min(Entiers)
4 0
5 >>> max(Entiers)
6 9
7 >>> sum(Entiers)
8 45
```

3.5 Méthodes

Le type de données *liste* possède des fonctions propres qui sont appelées méthodes en `Python`. Pour utiliser la méthode `method()` avec la liste nommée `list_name`, on utilise la syntaxe `list_name.method()`. Les méthodes associées aux listes sont indiquées dans le tableau 1.

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

Nom	Fonction
`append(x)`	Ajoute l'élément `x` en fin de liste
`extend(L)`	Ajoute la liste `L` en fin de liste
`insert(i,x)`	Ajoute l'élément `x` à la position `i`
`remove(x)`	Retire le premier élement de la liste de valeur `x`
`pop([i])`	Retire l'élément à la position `i` (ou le dernier élément si `i` n'est pas spécifié)
`index(x)`	Retourne l'indice du premier élément dont la valeur est `x`
`count(x)`	Compte les occurences de l'élément `x`
`sort()`	Trie les éléments
`reverse()`	Renverse l'ordre des éléments

TABLE 1 – Méthodes associées aux listes

4 Structures de contrôles

4.1 Conditions

Les tests, ou instructions conditionnnelles, sont un élément essentiel à tout langage informatique car ils permettent de prendre des décisions. En `Python`, les mots-clefs `if` et `elif` s'utilisent pour spécifier des conditions précises ainsi que des *blocs* d'instructions pour chaque condition. Le mot-clef `else` permet d'éxécuter un bloc d'instruction lorsque toutes les conditions associées aux mots-clefs `if` et `elif` sont fausses. La syntaxe générale pour un test à plusieurs cas est la suivante :

```
1 >>> if condition_1:
2 >>>     instructions_1
3 >>> elif condition_2:
4 >>>     instructions_2
5 >>> elif condition_3:
6 >>>     instructions_3
7 >>> else:
8 >>>     instructions_4
```

Nous pouvons noter la présence des deux-points à la fin de la ligne d'en-tête et l'indentation (décalage) du bloc des instructions. Voici un exemple de test à un cas qui est faux car la variable `x` n'est pas initialisée avec la valeur utilisée dans la condition du test.

```
1 >>> x = "souris"
2 >>> if x == "tigre":
3 >>>     print("Le test est vrai !")
```

Les tests multiples permettent de tester plusieurs conditions en même temps en utilisant les opérateurs booléens `and` et `or` qui s'écrivent en minuscule.

```
1 >>> x = 2
2 >>> y = 2
3 >>> if x == 2 and y == 2:
4 >>>     print("le test est vrai")
5 le test est vrai
```

Dans cet exemple, le même résultat peut être obtenu avec deux instructions `if` imbriquées.

```
>>> x = 2
>>> y = 2
>>> if x == 2:
>>>     if y == 2:
>>>         print("le test est vrai")
le test est vrai
```

Il est également possible de chainer les comparateurs.

```
>>> a, b, c = 1, 10, 100
>>> a < b < c
True
>>> a > b < c
False
```

4.2 Boucles

Le principe des boucles est de répéter plusieurs fois une série d'instructions. Chaque série d'instructions s'appelle une *itération*. Il existe deux types de boucles selon que l'on connait ou ignore le nombre d'itérations. Les boucles `for` répètent les instructions pour un nombre d'itérations fixé à l'avance tandis que les boucles `while` exécutent les instructions tant qu'une condition est vérifiée. La syntaxe des boucles comprend une en-tête précisant les conditions d'itérations et un bloc d'instructions constituant le corps de la boucle.

La syntaxe d'une boucle `for` est la suivante :

```
>>> for i in Liste:
>>>     instructions
```

où `i` désigne la variable d'itération qui prend successivement les différentes valeurs de la liste `liste`. Comme pour les tests, la ligne d'en-tête se termine par deux-points et le bloc des instructions est indenté. La fonction `range()` permet de faire une boucle sur une liste d'entiers de manière automatique. Il est possible d'itérer sur un sous-ensemble de liste.

```
>>> for i in [1, 2]:
>>>     print(i)
1
2

>>> for i in range(1,3):
>>>     print(i)

>>> for animal in Animaux[1:3]:
>>>     print(animal)
'tigre'
'singe'
```

La syntaxe d'une boucle `while` est la suivante :

```
>>> while condition:
>>>     instructions
```

où `condition` s'exprime à l'aide des opérateurs de comparaison. Le corps de la boucle est répété tant que `condition` est vraie. Si `condition` est fausse à l'initialisation de la boucle, le corps de la boucle n'est jamais exécuté. A l'inverse, si `condition` est toujours vraie, le corps de la boucle est répété indéfiniment. Comme pour une boucle `for`, il est nécessaire de mettre deux-points à la fin de la ligne d'en-tête et d'indenter le bloc des instructions.

```
>>> i=1
>>> while i<3:
>>>     print(i)
>>>     i += 1
1
2
```

L'instruction `break` permet de stopper l'exécution d'une boucle (`for` ou `while`) lorsqu'une condition externe est déclenchée. Cette instruction est généralement placée après une instruction conditionnelle `if`.

```
1 >>> i = 0
2 >>> for i in range(5):
3 >>>     if i == 3:
4 >>>         break
5 >>>     print(f'Le Nombre est {i}')
6 >>> print('En dehors de la boucle')
7 Le Nombre est 0
8 Le Nombre est 1
9 Le Nombre est 2
10 En dehors de la boucle
```

L'instruction `continue` permet d'éviter une partie d'une boucle lorsqu'une condition externe est déclenchée, mais de continuer pour terminer le reste de la boucle. Autrement dit, l'itération actuelle de la boucle est interrompue mais le programme revient au sommet de la boucle. Comme pour l'instruction `break`, l'instruction `continue` est généralement après une instruction conditionnelle `if`.

```
1 >>> i = 0
2 >>> for i in range(5):
3 >>>     if i == 3:
4 >>>         continue # continue ici
5 >>>     print(f'Le Nombre est {i}')
6 >>> print('En dehors de la boucle')
7 Le Nombre est 0
8 Le Nombre est 1
9 Le Nombre est 2
10 Le Nombre est 4
11 En dehors de la boucle
```

5 Modules

5.1 Définition

Dans les sections précédentes, nous avons rencontré des fonctions : `type()`, `len()`, `int()`, `float()`, `str()`, `range()`, `list()`, `min()`, `max()` et `sum()`. Les fonctions intégrées par défaut au langage `Python` sont relativement peu nombreuses et limitées à celles susceptibles d'être utilisées très fréquemment. Cependant, il existe beaucoup de fonctions déjà programmées en `Python` et regroupées dans des fichiers séparés appelés modules. Un module est un programme, implémenté par les développeurs de `Python`, contenant des fonctions. Un bon réflexe pour un programmeur est de vérifier les modules existants avant de commencer un code. Les modules aident également à structurer les développements puisqu'il est possible de créer ses propres modules en écrivant dans des fichiers les définitions et les instructions que l'on souhaite réutiliser.

5.2 Importation

Concrètement, un module est un fichier contenant des définitions et des instructions. Le nom de fichier du module est le nom du module suffixé de `.py`. Pour utiliser un module, il faut l'importer avec le mot-clef `import`. Pour accéder à une fonction d'un module, on utilise la syntaxe `module_name.function_name()` où `module_name` et `function_name()` désignent les noms respectifs du module et de la fonction. Dans l'exemple ci-dessous, la ligne 1 donne accès à toutes les fonctions du module `random` et la ligne 2 utilise la fonction `randint` pour renvoyer un nombre entier tiré aléatoirement entre 0 et 10 inclus.

```
1 >>> import random
2 >>> random.randint(0,10)
3 4
```

À l'aide du mot-clef `from`, on peut importer directement une fonction d'un module. Il est inutile de répéter le nom du module dans ce cas, seul le nom de la fonction est nécessaire.

```
1 >>> from random import randint
2 >>> randint(0,10)
3 7
```

On peut également importer toutes les fonctions d'un module en ajoutant $*$ après `import`.

```
1 >>> from random import *
2 >>> randint(0,50)
3 46
4 >>> uniform(0,1)
5 0.346270307583
```

Ceci étant, pour optimiser la mémoire et garder visible le nom du module, il est conseillé de charger le module seul puis d'appeler uniquement les fonctions utilisées. La personnalisation d'un nom de module avec un alias (nom plus court) est possible grâce au mot-clef `as`.

```
1 >>> import random as rand
2 >>> rand.randint(1,10)
3 6
```

Enfin, la fonction `help()` permet d'obtenir une description du module tandis que l'instruction `del` vide la mémoire d'un module déjà chargé.

```
1 >>> import random
2 >>> help(random)
...
n >>> del random
```

5.3 Principaux modules

Le tableau 2 indique quelques modules `Python` dont la liste exhaustive est disponible à la page `https://docs.python.org/fr/3/py-modindex.html`.

Nom	Fonction
`math`	Fonctions et constantes mathématiques de base (sin, cos, exp, pi, ...)
`cmath`	Fonctions mathématiques pour les nombres complexes
`random`	Génération de nombres (pseudo-)aléatoires
`statistics`	Fonctions statistiques sur des données numériques
`sys`	Interactions avec l'interpréteur `Python`
`os`	Dialogue avec le système d'exploitation
`time`	Accès à l'heure de l'ordinateur et aux fonctions gérant le temps
`urllib`	Récupération de données sur internet depuis `Python`

TABLE 2 – Exemples de modules `Python`

6 Fonctions

6.1 Définition

En programmation, l'intérêt des fonctions est d'une part d'optimiser les développements en évitant de répéter plusieurs fois les mêmes instructions et d'autre part de structurer le code en blocs logiques ce qui apporte lisibilité et clarté. Une fonction, appelée parfois routine, est caractérisée par trois éléments : i) une liste d'arguments d'entrées (optionnelle et éventuellement vide), ii) une série d'instructions définissant la fonction, iii) une liste de sorties (éventuellement vide).

Par exemple, la fonction `len()` prend en argument une liste et renvoie la longueur de cette liste. En revanche, la méthode `append(x)` (une méthode est une fonction associée à un objet) ajoute l'élément `x` à une liste mais ne renvoie rien. De même, la méthode `sort()` trie les éléments d'une liste sans sortie mais avec un argument optionnel.

```
1 >>> list = [3 ,2 ,1]
2 >>> len(list)
3 3

4 >>> list.append(0)
5 >>> print(list)
6 [3 ,2 ,1, 0]

7 >>> list.sort()
8 >>> print(list)
9 [0 ,1 ,2, 3]

10 >>> list.sort(reverse=True)
11 >>> print(list)
12 [3, 2, 1, 0]
```

6.2 Implémentation

Pour écrire une fonction en `Python`, on utilise le mot-clef `def` ainsi que le mot-clef `return` si l'on souhaite renvoyer une sortie. Comme pour les tests et les boucles, la ligne d'en-tête de la fonction se termine par deux-points et le bloc des instructions est indenté.

```
1 >>> def function_name(inputs):
2 >>>     Instructions
3 >>>     return output
```

```
1 >>> def somme(x,y):
2 >>>     s=x+y
3 >>>     return s
```

Il est essentiel de bien comprendre le comportement des variables dans la manipulation des fonctions. Lorsqu'une fonction est appelée, un espace de noms dédié à la fonction est réservé en mémoire. Cet espace est à distinguer de l'espace de noms où se trouvent les variables du programme principal. Les variables définies dans le programme principal sont appelées *variables globales* et sont visibles partout dans le programme. Les variables créées à l'intérieur d'une fonction sont appelées *variables locales* et sont connues uniquement lors de l'exécution fonction.

```
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 's' is not defined
```

Lors de la définition d'une fonction, il est possible de définir un argument par défaut pour chaque paramètre. Une fonction peut ainsi être appelée avec un sous-ensemble des arguments.

```
1 >>> def somme(x,y=2):
2 >>>     s=x+y
3 >>>     return s
```

```
1 >>> somme(10,10)
2 20
3 >>> somme(10)
4 12
```

7 Fichiers

7.1 Description

L'utilisation des fichiers est inévitable en programmation informatique pour diverses raisons. Tout d'abord, les fichiers permettent de gérer et de faire évoluer des données, même de petite taille. Ensuite, les fichiers aident à structurer les développements notamment en séparant les programmes qui les traitent. Enfin, les fichiers sont indispensables pour échanger des données avec d'autres programmes (peut-être écrits dans d'autres langages) et programmeurs. On peut voir des similarités entre l'utilisation d'un fichier et celle d'un livre : on les trouve à l'aide de leur titre, les ouvre et les referme. Tant que les livres et les fichiers sont ouverts, on peut y lire des informations diverses et y écrire des annotations. Les numéros de pages et de lignes permettent de se situer et l'on peut les consulter dans l'ordre ou le désordre.

Un fichier se compose de données enregistrées sur un disque dur, une disquette, une clef USB ou un CD/DVD. On y accède grâce à son nom qui peut inclure un nom de répertoire. En `Python`, le nom du répertoire courant s'obtient avec la fonction `getcwd()` (acronyme de get current working directory) du module `os`. Il est possible de changer le répertoire courant avec la méthode `chdir()` qui remplace le chemin actuel par le chemin indiqué entre parenthèses.

```
1 >>> import os
2 >>> rep_cour = os.getcwd()

3 >>> path = "new_path"
4 >>> os.chdir(path)
5 >>> new_rep_cour = os.getcwd()
```

En `Python`, l'accès aux fichiers est assuré par l'intermédiaire d'un objet-fichier créé avec la fonction intégrée `open()`. La syntaxe de cette fonction est `open('file_name','mode_name')` où le premier argument désigne le nom du fichier et le second argument est le mode d'ouverture spécifié par une lettre dont les différentes valeurs possibles sont indiquées dans le tableau 3. On peut assurer la lecture et l'écriture pour chaque mode en ajoutant `+` à lettre du mode. Le type de fichier peut être précisé : binaire ou texte avec les lettres `b` ou `t`. L'argument `'r+t'` permet par exemple d'avoir accès, en lecture et en écriture, à un fichier texte existant.

Lettre	Mode
`'r'`	Accès en lecture seule (mode par défaut). Le fichier doit exister.
`'w'`	Accès en écriture seule. Si le fichier n'existe pas, il est créé. Sinon, il est vidé.
`'x'`	Accès en écriture, mais exclusivement en création. Le fichier ne doit pas exister.
`'a'`	Accès en écriture seule en fin du fichier. Si le fichier n'existe pas, il est créé.
`'r+'`	Mode `r` avec écriture
`'w+'`	Mode `w` avec lecture
`'x+'`	Mode `x` avec lecture
`'a+'`	Mode `a` avec lecture
`'b'`	Mode binaire
`'t'`	Mode texte (par défaut)

TABLE 3 – Second argument de la fonction `open`

7.2 Lecture

Nous allons illustrer les options de lecture avec le fichier texte `zoo.txt` de 4 lignes.

```
1 girafe
2 tigre
3 singe
4 souris
```

Dans l'exemple ci-dessous, la ligne 1 ouvre le fichier en lecture seule (sans le lire). L'instruction `open('zoo.txt','r')` suppose que le fichier `zoo.txt` est dans le répertoire depuis lequel l'interpréteur `Python` a été lancé. Si ce n'est pas le cas, il faut préciser le chemin d'accès au fichier. La variable `obFichier` est un objet de type fichier ouvert (lignes 2 et 3). Ensuite, la méthode `readlines()` appliquée à `obFichier` permet de lire le fichier puis d'afficher son contenu sous forme d'une liste contenant toutes les lignes du fichier (lignes 4 et 5). La méthode `close()`, sans rien renvoyer, modifie l'état de l'objet (ligne 6) qui est désormais fermé comme l'illustrent le message d'erreur des lignes 7 à 10 renvoyé suite à une nouvelle tentative de lecture.

```
1 >>> obFichier = open('zoo.txt','r')
2 >>> obFichier
3 <open file 'zoo.txt', mode 'r' at 0x7f7d4dea5c90>

4 >>> obFichier.readlines()
5 ['girafe\n', 'tigre \n', 'singe\n', 'souris\n']

6 >>> obFichier.close ()
7 >>> obFichier.readlines()
8 Traceback (most recent call last):
9 File "<stdin>", line 1, in <module>
10 ValueError: I/O operation on closed file
```

En `Python`, le mot-clef `with` regroupe les étapes de d'ouverture et de fermeture. Le bloc d'instructions suivant ce mot-clef doit être indenté.

```
1 >>> with open ('zoo.txt','r') as obFichier:
2 >>>     for ligne in obFichier:
3 >>>         print(ligne)
```

D'autres méthodes que `.readlines()` existent pour lire et manipuler un fichier. La méthode `.read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

```
1 >>> x=obFichier.read()
2 >>> print(x)
3 ['girafe\ntigre\nsinge\nsouris\n']
```

La méthode `.readline()` (sans `s` à la fin) lit une ligne d'un fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de `.readline()`, la ligne suivante est renvoyée. Cette méthode permet de lire un fichier ligne par ligne en utilisant une boucle `while`.

```
1 >>> ligne = obFichier.readline()
2 >>> while ligne != "":
3 >>>     print(ligne)
4 >>>     ligne = obFichier.readline()
5 girafe
6
7 tigre
8
9 singe
10
11 souris
```

Des itérations sur l'objet-fichier offrent également un moyen simple pour parcourir un fichier.

```
1 >>> for ligne in obFichier:
2 >>>     print(ligne)
3 girafe
4
5 tigre
6
7 singe
8
9 souris
```

7.3 Ecriture

L'écriture dans un fichier s'effectue en appliquant la fonction intégrée `write()` à l'objet-fichier. Dans l'exemple ci-dessous, une liste de trois animaux est créée (ligne 1). Ensuite, nous ouvrons (et créons) en mode écriture un fichier intitulé `'zoo2.txt'` dont l'objet-fichier associé se nomme `obFichier2` (ligne 2). L'écriture dans le fichier se fait en parcourant la liste d'animaux avec une boucle `for` (lignes 3 et 4). Le fichier `zoo2.txt` contient la chaîne de caractère `poissonabeillechat`. Pour obtenir un nom d'animal par ligne, il faut ajouter le caractère de fin de ligne `'\n'` après chaque nom d'animal (ligne 4b).

```
1 >>> animaux2 = ["poisson","abeille","chat"]
2 >>> with open ("zoo2.txt","w") as obFichier2:
3 >>>     for i in animaux2:
4 >>>         obFichier2.write(i)

3b >>>    for i in animaux2:
4b >>>        obFichier2.write(i+'\n')
```

8 Bibliothèques scientifiques

8.1 NumPy

La bibliothèque `NumPy`, abréviation de Numerical Python (`http://www.numpy.org/`), est destinée à la manipulation de tableaux numériques multidimensionnels pour effectuer des calculs numériques. Les tableaux sont créés avec la commande `np.array(data,type)`. Le premier argument contient les coefficients du tableau écrits entre crochets et le second indique le type des variables qui est identique pour tous les éléments (contrairement aux listes qui peuvent contenir des éléments ayant des types différents). Une liste de coefficients permet de définir un vecteur et une liste de liste de coefficients permet de définir une matrice.

```
1 >>> import numpy as np
2 >>> vec = np.array([1,2,3,4],float)
3 >>> vec
4 array([1., 2., 3., 4.])

5 >>> tab=np.array([[1,2],[3,4],[5,6]])
6 >>> tab
7 array([[1, 2],
8        [3, 4],
9        [5, 6]])
```

Les vecteurs peuvent être définis avec la fonction `np.array(range())` pour les entiers ou les fonctions `np.linspace()` et `np.arange()` plus généralement. Il existe des fonctions pour créer des matrices nulles, constantes, identités, diagonales, bandes et aléatoires. Les méthodes `size` et `shape` renvoient la taille des vecteurs et matrices.

```
1 >>> u = np.array(range(a,b))       # Vecteur d'entiers entre a et b
2 >>> v = np.array(range(a,b,i))     # Vecteur d'entiers entre a et b avec pas de i
3 >>> w = np.linspace(a,b,N)         # Vecteur de N reels entre a et b avec pas cst
4 >>> x = np.arange(a,b,f)           # Vecteur de reels entre a et b avec pas de f

5 >>> Z = np.zeros([n,p])            # Matrice nulle de taille nxp
6 >>> U = np.ones([n,p])             # Matrice de 1 de taille nxp
7 >>> I = np.eye(n)                  # Matrice identite de taille n
8 >>> D = np.diag(d)                 # Matrice diagonale de coefficients d
9 >>> B = np.diag(d,1)               # Matrice bande de sur-diagonale d
10 >>> C = np.diag(d,-1)             # Matrice bande de sous-diagonale d
11 >>> R = np.random.rand(n,p)       # Matrice aleatoire (entre [0,1]) de taille nxp

12 >>> u.size                        # Taille d'un vecteur
13 >>> U.shape                       # Taille d'une matrice
```

Les syntaxes pour accéder aux coefficients (et éventuellement les modifier) d'un tableau sont proches de celles utilisées avec les listes.

```
>>> M = np.random.rand(10,10)
>>> M[i,j]                          # Coefficient i,j
>>> M[i,:]                          # Ligne i
>>> M[:,j]                          # Colonne j
>>> M[i1:i2,j1:j2]                  # Sous-matrice bloc, i1<=i<i2 et j1<=j<j2
```

Plusieurs opérations sont possibles sur les vecteurs et les matrices.

```
>>> u = np.random.rand(10,1)
>>> A = np.random.rand(10,2)
>>> u**3, A**3                      # Mise a la puissance n=3
>>> 1./u, 1./A                      # Tableaux des inverses des coefficients
>>> np.transpose(u)                 # Transposee
>>> np.trace(A)                     # Trace
>>> np.sum(u)                       # Somme
>>> np.prod(u)                      # Produit
>>> np.mean(u)                      # Moyenne
>>> np.min(u)                       # Minimum
>>> np.max(u)                       # Maximum
>>> np.argmin(u)                    # indice du minimum
>>> np.argmax(u)                    # indice du maximum
>>> np.sum(A,axis=d)                # Somme dans la direction d
>>> A.reshape(new_shape)            # Redimensionnement
>>> A.flatten()                     # Redimensionnement en un vecteur
```

Plusieurs opérations sont possibles entre les vecteurs et les matrices ; certaines de ces opérations sont soumises à des conditions sur la taille des différents tableaux.

```
>>> v = np.random.rand(10,1)
>>> B = np.random.rand(10,2)
>>> u+v, A+B                        # Addition terme a terme
>>> u-v, A-B                        # Soustraction terme a terme
>>> u*v, A*B                        # Produit terme a terme
>>> u/v, A/B                        # Division terme a terme
>>> np.vdot(u,v)                    # Produit scalaire
>>> np.outer(u,v)                   # Produit externe
>>> np.dot(np.transpose(A),B)       # Produit matriciel
>>> np.concatenate((u,v),axis=d)    # Concatenation dans la direction d
```

Le module `linalg` fournit des fonctions pour calculer le rang, le déterminant, la norme, le conditionnement, l'inverse d'une matrice. Il existe également des fonctions pour calculer les éléments propres d'une matrice et résoudre des systèmes linéaires.

```
>>> np.linalg.matrix_rank(R)        # Rang
>>> np.linalg.det(R)                # Determinant
>>> np.linalg.norm(R,p)             # Norme p=1,2,'inf' ou 'fro'
>>> np.linalg.cond(R,p)             # Conditionnement en norme p
>>> np.linalg.inv(R)                # Inverse
>>> np.linalg.pinv(R)               # Pseudo-Inverse
>>> val, vec = np.linalg.eig(A)     # Valeurs et vecteurs propres
>>> x = np.linalg.solve(A,b)        # Resolution de Ax=b
```

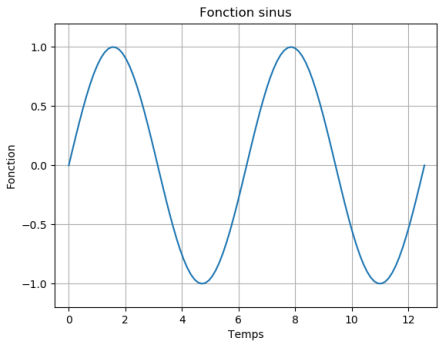
`Numpy` dispose de nombreuses fonctions mathématiques.

```
>>> np.sin(x)                       # Sinus
>>> np.cos(x)                       # Cosinus
>>> np.tan(x)                       # Tangente
>>> np.arcsin(x)                    # ArcSinus
>>> np.sinh(x)                      # Sinus hyperbolique
>>> np.arcsinh(x)                   # Inverse du Sinus hyperbolique
>>> np.exp(x)                       # Exponentielle
>>> np.log(x), np.log10(x)          # Logarithme
>>> np.sqrt(x)                      # Racine carree
```

8.2 Matplotlib

`Matplotlib` est une bibliothèque permettant de tracer une très grande variété de graphiques (`http://matplotlib.org/gallery.html`)). Cette bibliothèque est un outil complémentaire aux bibliothèques `NumPy` (cf. section précédente) et `Scipy` (cf. section suivante). L'alias communément utilisé pour `Matplotlib` est `plt`. Chaque nouvelle figure commence avec `plt.figure()` et l'affichage s'effectue avec `plt.show()` qui ouvre une fenêtre pour chaque figure créée. La commande de base est `plt.plot` (ligne 6 de l'exemple ci-après) qui peut être enrichie par de nombreuses commandes notamment un titre pour la figure ainsi que le nom et les limites de chaque axe.

```
1  >>> import numpy as np
2  >>> import matplotlib.pyplot as plt
3  >>> x = np.linspace(0,4*np.pi,100)
4  >>> y = np.sin(x)
5  >>> plt.figure()                    # Creation figure
6  >>> plt.plot(x, y)                  # Courbe
7  >>> plt.title('Fonction sinus')     # Titre de la figure
8  >>> plt.xlabel('Temps')             # Nom axe x
9  >>> plt.ylabel('Fonction')          # Nom axe y
10 >>> plt.xlim([-0.5, 13])            # Limites axe x [xmin, xmax]
11 >>> plt.ylim([-1.2, 1.2])           # Limites axe y [ymin, ymax]
12 >>> plt.grid()                      # Grille
13 >>> plt.show()                      # Affichage graphe
```



La fonction `plt.plot(x, y,options)` possède de nombreux arguments facultatifs. La couleur, le style et les symboles d'une courbe peuvent être renseignés de manière condensée avec une chaîne de caractère. Les principales valeurs des trois paramètres précédents sont indiquées dans les tableaux 4, 5 et 6. Un ou plusieurs de ces paramètres peuvent être indiqués dans un ordre arbitraire, les chaînes `'b-'` et `'-b'` traceront par exemple une courbe avec une ligne continue bleue tandis que les chaînes `'r-.o'`, `'ro-.'`, `'-.ro'`, `'-.or'`, `'or-.'` et `'o-.r'` traceront une courbe avec une ligne pointillée et des points rouges. Une description plus complète est possible grâce à plusieurs mots-clefs comme `color`, `linewidth`, `linestyle`, `marker`, `markersize`, `markerfacecolor`, `markeredgecolor`, `markeredgewidth`, etc, dont les noms sont suffisamment explicites.

Chaîne	Couleur
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

Table 4 – Couleurs

Chaîne	Liaison
`-`	ligne continue
`--`	tirets
`:`	ligne pointillée
`-.`	tirets points

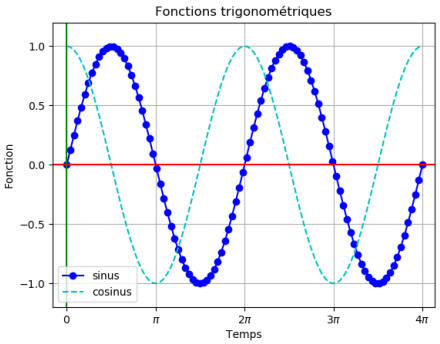
Table 5 – Liaisons

Chaîne	Symbole
.	point
,	pixel
o	rond
s	carré
p	pentagone
+	plus
×	croix
*	étoile

Table 6 – Symboles

Il est possible de tracer plusieurs courbes sur le même graphique en utilisant des styles différents ainsi qu'une légende en utilisant la fonction `plt.legend([data1,data2])`. Les instructions `plt.xlim()` et `plt.ylim()` peuvent être regroupées dans l'unique instruction `plt.axis()`. Les instructions `plt.xticks()` et `plt.yticks()` indiquent des marques et noms sur les axes. Ces derniers peuvent être explicitement tracés avec les instructions `plt.axhline()` et `plt.axvline()`.

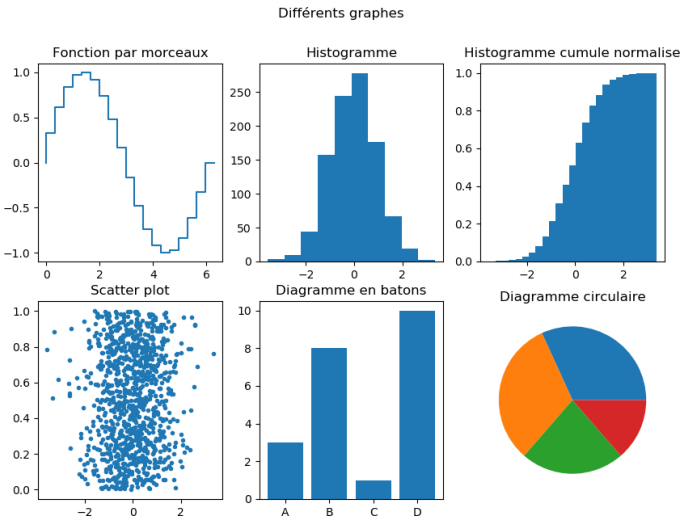
```
>>> x = np.linspace(0,4*np.pi,100)
>>> Pi = np.pi
>>> plt.figure()                                    # Creation figure
>>> plt.plot(x, np.sin(x),'b-o')                    # Courbe 1
>>> plt.plot(x, np.cos(x),'c--')                    # Courbe 2
>>> plt.xlabel('Temps')                             # Nom axe x
>>> plt.ylabel('Fonction')                          # Nom axe y
>>> plt.title('Fonctions trigonometriques')         # Titre de la figure
>>> plt.axis([-0.5, 13, -1.2, 1.2])                 # Cadre [xmin, xmax, ymin, ymax]
>>> plt.legend(['sinus','cosinus'])                 # Legende
>>> plt.xticks([0,Pi, 2*Pi, 3*Pi,4*Pi],            # Marques et noms sur axe x
        [r'$0$', r'$\pi$', r'$2\pi$', r'$3\pi$', r'$4\pi$'])
>>> plt.axhline(color='r')                          # Axe x en rouge
>>> plt.axvline(color='g')                          # Axe y en vert
>>> plt.grid()                                      # Grille
>>> plt.show()                                      # Affichage graphe
```



Il existe diverses types de tracés graphiques : `plt.step()` pour les fonctions par morceaux, `plt.hist()` pour les histogrammes, `plt.scatter()` pour les nuages de points, `plt.bar()` pour les diagrammes en bâtons, `plt.pie()` pour les diagrammes circulaires, etc. Par ailleurs, il est possible de regrouper plusieurs graphiques dans une même figure avec la commande `subplot(p,q,n)` où `(p,q)` représente la taille du tableau des sous-graphiques et `n` la position dans ce tableau. Si $p = 2$ et $q = 3$, une grille de 6 graphiques (2 lignes et 3 colonnes) numérotés de 1 à 6 est créée.

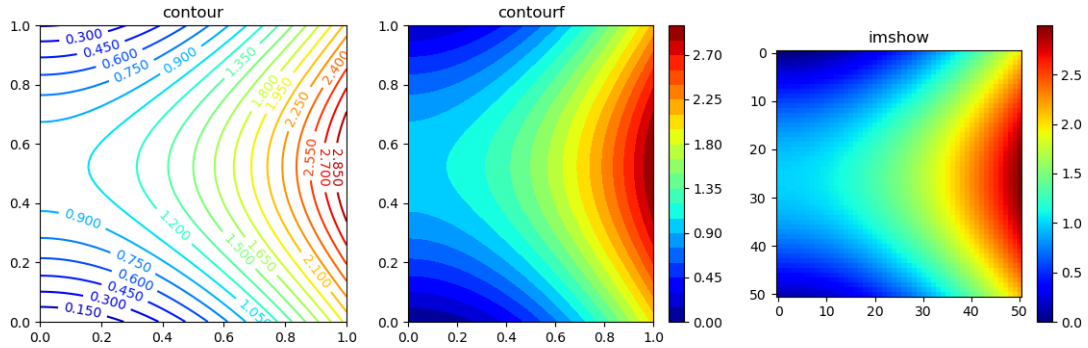
```
>>> fig = plt.figure(figsize=(10,7))
>>> plt.suptitle("Differents graphes")              # Titre general

>>> plt.subplot(2,3,1)                              # Grille 3x2 - Graphique 1
>>> x = np.linspace (0,2*np.pi,20)
>>> y = np.sin(x)
>>> plt.step(x,y)
>>> plt.title('Fonction par morceaux')

>>> plt.subplot(2,3,2)                              # Graphique 2
>>> x=np.random.randn(1000)
>>> y=np.random.rand(1000)
>>> plt.hist(x)
>>> plt.title('Histogramme')

>>> plt.subplot(2,3,3)                              # Graphique 3
>>> plt.hist(x,25,density=True,cumulative=True)
>>> plt.title('Histogramme cumule normalise')

>>> plt.subplot(2,3,4)                              # Graphique 4
>>> plt.title('Scatter plot')
>>> plt.scatter(x,y,marker='.')
```

```
>>> plt.subplot(2,3,5)                          # Graphique 5
>>> X = np.array(["A","B","C","D"])
>>> Y = np.array([3,8,1,10])
>>> plt.bar(X,Y)
>>> plt.title('Diagramme en batons')

>>> plt.subplot(2,3,6)                          # Graphique 6
>>> z = np.array([35,35,25,15])
>>> plt.pie(z)
>>> plt.title('Diagramme circulaire')

>>> plt.show()
```



`Matplotlib` comporte aussi des fonctions pour représenter des données en dimensions supérieures. La fonction `np.meshgrid()` qui génère des grilles cartésiennes (potentiellement irrégulières) est particulièrement efficace pour traiter des données structurées. On trace par exemple les lignes de niveaux d'une fonctions de deux variables avec la fonction `contour()` et sa version remplie `contourf()`. On peut aussi utiliser la fonction `matshow()` dédiée à la représentation des matrices. Pour terminer, la commande `plt.savefig('filename')` sauvegarde la figure courante dans le fichier `filename`.

```
>>> N=50                                        # Nb points
>>> x=np.linspace (0,1,N+1)                     # Vecteur x
>>> y=np.linspace (0,1,N+1)                     # Vecteur y
>>> X,Y=np.meshgrid(x,y)                        # Grilles X et Y
>>> Z=2*X*X+np.sin (3*Y)                        # Z(X,Y)

>>> plt.figure(figsize=(10,4))
>>> plt.set_cmap('jet')                         # Color map

>>> plt.subplot (1,3,1)                         # Figure 1
>>> c=plt.contour(X,Y,Z,20)                     # 20 lignes de niveaux
>>> plt.clabel(c)                               # Etiquettes
>>> plt.title("contour")                        # Titre

>>> plt.subplot (1,3,2)                         # Figure 2
>>> plt.contourf(X,Y,Z,20)                      # Remplissage avec 20 niveaux
>>> plt.colorbar()                              # Barre laterale
>>> plt.title("contourf")                       # Titre

>>> plt.subplot (1,3,3)                         # Figure 3
>>> plt.imshow(Z)                               # Remplissage matrice
>>> plt.colorbar()                              # Barre laterale
>>> plt.title("imshow")                         # Titre

>>> plt.savefig('Image.png')                    # Sauvegarde
>>> plt.show ()
```



8.3 SciPy

La bibliothèque `Scipy`, abréviation de Scientific Python (`https://scipy.org/`), est dédiée au calcul scientifique. Cette bibliothèque est basée sur `NumPy` et fournit des outils et des algorithmes standards répartis dans un ensemble de modules spécifiques dont les principaux sont indiqués dans le tableau 7.

Nom	Domaine
`fft`	Transformées de Fourier
`integrate`	Intégration numérique et solveurs d'ode
`interpolate`	Interpolation (univariée et multivariée)
`io`	Lecture et écriture de tableaux numériques
`linalg`	Algèbre linéaire
`optimize`	Optimisation (avec et sans contraintes)
`signal`	Traitement du signal
`special`	Fonctions spéciales (Bessel, gamma, etc).
`sparse`	Matrices creuses et systèmes linéaires
`stats`	Statistiques

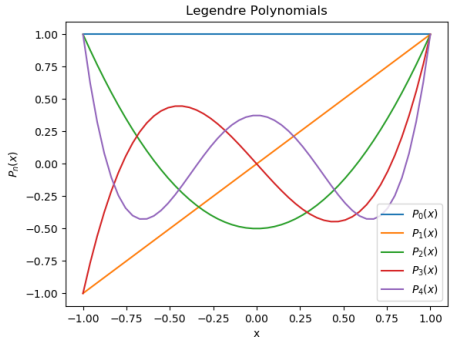
TABLE 7 – Principaux modules de `Scipy`

○ Fonctions spéciales

De nombreuses fonctions, utiles notamment en physique, sont disponibles dans le module `scipy.special`. Ce module contient par exemple i) les fonctions d'Airy, de Bessel, Gamma, Beta, ii) les fonctions et intégrales elliptiques, iii) de nombreux polynômes orthogonaux comme ceux de Legendre, Hermite, Chebyshev, Jacobi, iv) la fonction d'erreur (et son inverse), v) les harmoniques sphériques, etc. Nous pouvons noter que les polynômes appartiennent à la classe `poly1d` de `NumPy`. La fonction `eval_legendre()` permet par exemple d'évaluer les polynômes de Legendre que l'on peut ensuite représenter.

```
1 >>> import numpy as np
2 >>> from scipy.special import gamma, erf, legendre
3 >>> gamma([0, 0.5, 1, 1.5])
4 array([inf, 1.77245385, 1., 0.88622693])

5 >>> erf([-10,-1,-0.5,0,0.5,1,10])
6 array([-1., -0.84270079, -0.52049988, 0., 0.52049988, 0.84270079,  1.])

7 >>> legendre(3)
8 poly1d([2.5, 0. ,-1.5, 0. ])
```

```
>>> from scipy.special import eval_legendre
>>> x=np.linspace(-1,1)

>>> plt.figure()
>>> for n in range(0,5):
>>>     y = eval_legendre(n,x)
>>>     plt.plot(x, y, label=r'\$P_{}(x)\$'.format(n))

>>> plt.title('Legendre Polynomials')
>>> plt.xlabel('x')
>>> plt.ylabel(r'\$P_n(x)\$')
>>>  plt.legend(loc='lower right')
>>>  plt.show()
```



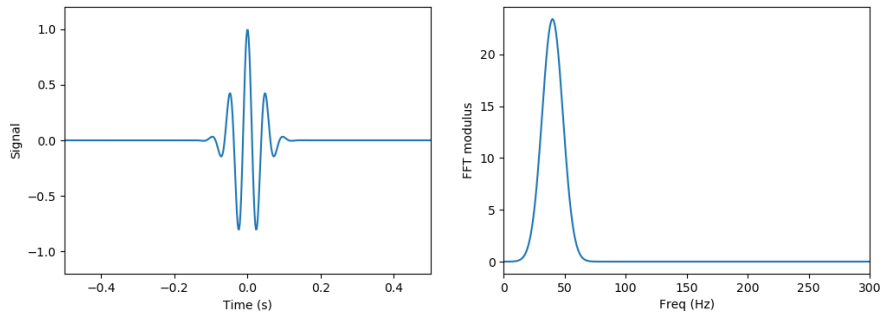
○ Algèbre creux

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix, csr_matrix
>>> row = np.array([0, 3, 1, 0])
>>> col = np.array([0, 3, 1, 2])
>>> coeff = np.array([4, 5, 7, 9])
>>> M1 = coo_matrix((coeff, (row,col)), shape=(4,4))
>>> M1
<4x4 sparse matrix of type '<class 'numpy.int64'>'
with 4 stored elements in COOrdinate format>

>>> M1.toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])

>>> M2 = csr_matrix((coeff, (row,col)), shape=(4,4))
>>> M1-M2
<4x4 sparse matrix of type '<class 'numpy.int64'>'
with 0 stored elements in Compressed Sparse Row format>

>>> M1.nnz
4

>>> M1.nonzero()
array([0, 3, 1, 0], dtype=int32), array([0, 3, 1, 2], dtype=int32))
```

○ Transformée de Fourier

Le module `fft` comporte des algorithmes de transformées de Fourier rapides (FFT) en dimension 1, 2 et n ainsi que leurs inverses. Ce module possède également des algorithmes de transformées en cosinus discrètes (de types I, II et III), de transformées de Hankel ainsi que pour le produit de convolution.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import fft
>>> from scipy.signal import gausspulse
>>> t = np.linspace(-1, 1, 1000)                # Intervalle de temps
>>> x = gausspulse(t, fc=20, bw=0.5)            # Signal
>>> F = fft.fft(x)                              # TFD
>>> freqs = fft.fftfreq(len(x),1./1000.)        # Frequences en Hz

>>> fig, axes = plt.subplots(1, 2, figsize=(12,4))

>>> axes[0].plot(t, x)                          # Signal
>>> axes[0].set_xlim([-0.5, 0.5])
>>> axes[0].set_ylim([-1.2, 1.2])
>>> axes[0].set_xlabel('Time (s)')
>>> axes[0].set_ylabel('Signal')

>>> ind = freqs>0
>>> axes[1].plot(freqs[ind], np.abs(F[ind]))    # Module de la TFD
>>> axes[1].set_xlim([0, 300])
>>> axes[1].set_xlabel('Freq (Hz)')
>>> axes[1].set_ylabel('FFT modulus')

>>> plt.show()
```



9 Web scraping

9.1 Définition

Le web scraping, appelé aussi harvesting (moissonnage), est une technique d'extraction automatisée de données provenant de diverses pages internet. Les données collectées au format `HTML` sont transformées en d'autres formats plus exploitables (`excel`, `csv`, $\cdots$) et l'objectif du web scraping est de réutiliser ces données dans un autre contexte comme l'enrichissement de bases de données, le référencement ou l'exploration de données. Le web scraping est à la base de nombreuses applications comme les moteurs de recherche et les comparateurs de prix. Il permet aussi de récupérer les contacts, adresse e-mail ou numéro de téléphone, de personnes qui ont renseigné ces informations sur des sites. La récupération de données librement diffusées n'est pas illégale mais leur réutilisation, telles quelles ou après transformation mineure, présente des risques et il peut être nécessaire de faire une étude juridique fine axée sur le niveau de transformation et de réutilisation des données collectées.

9.2 Language HTML

Le contenu d'une page web et sa structure sont codés en HTML qui signifie HyperText Markup Language et se traduit par « langage de balises pour l'hypertexte ». Le language HTML est constitué d'un ensemble de commandes qui indiquent comment "mettre en page" un texte. Il s'agit essentiellement de définitions logiques précisant le type de texte comme titre, sous-titre, liste, etc. Chaque type est défini par un environnement délimité par des balises et la syntaxe est

```
<tag>
Contenu de l'environnement
</tag>
```

où tag est spécifique à chaque environnement. Plusieurs balises sont utilisées dans l'exemple ci-après notamment les balises head et body qui permettent de définir les informations cachées et celles visibles du site. La balise head permet de donner des informations sur le site comme son titre, défini par la balise title, et des mot-clefs que les moteurs de recherche peuvent exploiter. Ces informations ne font pas parties du corps du document contrairement aux informations définies par la balise body. Les balises h1, h2 et h3 permettent de définir des titres de niveau 1, 2 et 3, la balise p des paragraphes, les balises ul et li des listes. La balise a avec l'attribut href crée un lien hypertexte vers une autre page, des adresses e-mail ou des fichiers. Les commandes HTML ne sont pas sensibles aux caractères minuscules ou majuscules et la plupart des balises peuvent être imbriquées mais pas chevauchées.

```
<!DOCTYPE html>
<html>
    <head>
        <title> Page HTML</title>
    </head>
    <body>
        <h1> Titre de niveau 1</h1>
        <p> Un paragraphe.</p>
        <h2> Titre de niveau 2</h2>
        Une liste
         <ul>
           <li> Méthodes numériques </li>
           <li> Quantification d'incertitudes </li>
           <li> Inférence Bayésienne </li>
         </ul>
       <h3> Titre de niveau 3</h3>
       Un lien <a href="https://www.estp.fr/campus-dorleans">ESTP Orleans</a>.
    </body>
</html>
```

L'affichage de ce code dans un navigateur donne

Titre de niveau 1

Un paragraphe.

Titre de niveau 2

Une liste

- Méthodes numériques
- Quantification d'incertitudes
- Inférence Bayésienne

Titre de niveau 3

Un lien ESTP Orleans.

9.3 Implémentation

Le web scraping comporte deux étapes : i) le téléchargement de la page à étudier et ii) l'analyse du code HTML associé. Le téléchargement du contenu d'une page web s'effectue par une requête HTTP (HyperText Transfer Protocol) qui peut s'effectuer en Python avec la fonction get du module requests.

```
>>> from requests import get
>>> url = "https://fr.wikipedia.org/wiki/Star_Wars"      # Adresse page
>>> response = get(url)                                  # Requete
>>> source = None                                        # Initialisation source
>>> if response.status_code == 200 :                     # Si succes
>>>     source = response.text                           # Code source
```

Lorsque le contenu de la page est téléchargé, on peut utiliser la bibliothèque BeautifulSoup qui permet d'analyser le contenu d'une page HTML. BeautifulSoup fournit des méthodes simples pour naviguer, rechercher et modifier un arbre d'analyse dans des fichiers HTML ou XML. Il transforme un document HTML complexe en un arbre d'objets Python. Le nom BeautifulSoup fait référence au terme « tag soup » (soupe aux balises) utilisé dans le domaine du développement web pour désigner péjorativement l'écriture en HTML. La commande fondamentale est de créer un object BeautifulSoup à partir du code source.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(source,'html.parser')
>>> type(soup)
bs4.BeautifulSoup
```

La méthode prettify() permet de visualiser le contenu du code source et title son titre. Plus généralement, la méthode find_all('tag') retourne la liste des environnements balisés par tag. La méthode .text.strip() permet d'extraire le texte sans les balises.

```
>>> print(soup.prettify())
<!DOCTYPE html> ...

>>> soup.title
<title> Star Wars - Wikipedia </title>

>>> soup.find_all('h2')
[<h2 class="vector-pinnable-header-label">Sommaire</h2>,
 <h2><span class="mw-headline" id="Univers">Univers</span></h2>,
<h2><span class="mw-headline" id="Films">Films</span></h2>,
<h2><span class="mw-headline" id="Critiques">Critiques</span></h2>,
<h2><span id="Univers_.C3.A9tendu"></span><span class="mw-headline" id="
                                        Univers_etendu">Univers etendu</span></h2
                                        >,
<h2><span class="mw-headline" id="Inspirations">Inspirations</span></h2>,
<h2><span id="Post.C3.A9rit.C3.A9"></span><span class="mw-headline" id="
                                        Posterite">Posterite</span></h2>,
<h2><span id="Notes_et_r.C3.A9f.C3.A9rences"></span><span class="mw-headline" id
                                        ="Notes_et_references">Notes et
                                        references</span></h2>,
<h2><span class="mw-headline" id="Annexes">Annexes</span></h2>]

>>> h2title = soup.find_all('h2')
>>> for tag in h2title:
>>>     print(tag.text.strip())
Sommaire
Univers
Films
Critiques
Univers etendu
Inspirations
Posterite
Notes et references
Annexes
```