Informatique
Programmer en Python

ECG

1ère année

Table des matières

Chapitre 1

Présentation et organisation des cours d'informatique

1.1 Ce que dit le programme

Les séances de travaux pratiques du premier semestre poursuivent les objectifs suivants :
- *consolider l'apprentissage de la programmation qui a été entrepris dans les classes du lycée en langage Python ;*
- *mettre en place une discipline de programmation : découpage modulaire à l'aide de fonctions et programmes, annotations et commentaires, évaluation par tests ;*
- *mettre en pratique des algorithmes facilitant le traitement de l'information, la modélisation, la simulation.*

1.2 Qu'est-ce qu'un programme informatique

Un **algorithme** est la description d'une suite d'étapes non ambiguës et en nombre ni permettant d'obtenir un résultat à partir d'éléments fournis en entrée. Par exemple, une recette de cuisine est un algorithme permettant d'obtenir un plat à partir de ses ingrédients !

Pour qu'un algorithme puisse être mis en œuvre par un ordinateur, il faut qu'il soit exprimé dans un langage compris par l'ordinateur.
Un **programme informatique** est la traduction d'un algorithme en instructions qui sont compréhensibles et exécutables par l'ordinateur. Comme l'ordinateur ne sait manipuler que du binaire (c'est-à-dire une succession de 0 et de 1), il est nécessaire d'utiliser un **langage de programmation** pour écrire de façon lisible (c'est-à-dire avec des instructions compréhensibles par l'humain car proches de son langage) les instructions que l'ordinateur devra exécuter.
Parmi les plus connus, nous pouvons citer C, C++, Java et Python que nous utiliserons pour les TP informatique.

Afin que ces instructions soient comprises par l'ordinateur, elles sont généralement traduites en langage machine (c'est à dire en binaire) par ce que l'on appelle un **compilateur** (c'est le traducteur). On distingue **les langages de programmation compilés** des langages **interprétés**. Les premiers ont pour eux de meilleures performances, mais il nécessitent d'être compilés sur chacune des machines où ils seront exécutés. Les langages interprétés sont généralement moins performants mais peuvent être exécutés directement sur une machine compatible, sans avoir besoin d'être compilé au préalable. Python est un langage de programmation interprété.

D'une façon générale, le programme est un simple fichier texte (écrit avec un traitement de texte ou un éditeur de texte), que l'on appelle **fichier source**. Le fichier source contient les lignes de programmes que l'on appelle **code source**.

La façon d'écrire un programme est intimement liée au langage de programmation que l'on a choisi. Ces langages de programmation sont généralement intégrés dans ce que l'on appelle un **logiciel de programmation** qui contient des programmes informatiques déjà définis et qui permet d'en construire de nouveau.

Chapitre 2

Introduction à Python

2.1 Histoire de Python

Python est un langage de programmation open source créé par le programmeur Guido van Rossum en 1991

Pour remonter à l'origine de Python, il faut revenir dans les années 1980 à l'institut national de recherche mathématiques et informatiques des Pays-Bas où un certain Guido van Rossum travaille sur le développement d'un nouveau langage de programmation, l'ABC, destiné à être un successeur du BASIC et du PASCAL. Contrairement aux précédents langages, l'imbrication du code y est déterminée par l'indentation des lignes (nous y reviendrons plus en détails par la suite). Cependant plusieurs contraintes de tailles empêchèrent à l'époque sa diffusion à un large public : lecture/écriture de fichiers difficiles, pas de concept de bibliothèque, système d'entrée/sortie peu souple.

Guido van Rossum est ensuite chargé de développer un système d'exploitation et de créer un langage de script pour le manipuler. S'inspirant de ses précédents travaux tout en rectifiant les inconvénients de l'ABC, il conçut les premières versions d'un langage qu'il appela Python en hommage aux Monty Python qu'il admirait.

Au fur et à mesure des années, ce langage de programmation s'est hissé parmi les plus utilisés dans le domaine du développement de logiciels et d'analyse de données. Il est aussi un des éléments moteur de l'expansion du Big Data.

2.2 Les avantages de Python

Python doit sa popularité à plusieurs avantages qui profitent aussi bien aux débutants qu'aux experts.

- **un langage libre et gratuit :** avec une communauté qui répond aux questions des utilisateurs de manière aussi rapide, efficace et précise qu'un support commercial et une documentation très complète et à jour ;
- **un langage interprété :** il ne nécessite donc pas d'être compilé pour fonctionner. Un programme interpréteur permet d'exécuter le code Python sur n'importe quel ordinateur. Ceci permet de voir rapidement les résultats d'un changement dans le code ;
- **un langage performant :** Python n'est pas un langage interprété classique. Le code source est en effet d'abord transformé lors de la première exécution en bytecode Python, plus proche du langage machine que le code source, et donc plus rapide lors des exécutions suivantes ;
- **un langage facile à apprendre et à utiliser :** sa syntaxe est conçue pour être directe et lisible, ses caractéristiques sont peu nombreuses, ce qui permet de créer des programmes rapidement et avec peu d'efforts et permet ainsi de se focaliser sur ce que l'on fait plutôt que sur la façon dont on le fait. Comme nous le verrons plus loin , pas de point virgule ou d'accolade pour indiquer la fin d'une instruction ou d'un bloc de code, c'est l'indentation qui fait la structure du programme et rend ainsi les programmes très faciles à lire, même pour un débutant ;

- un langage populaire : Python fonctionne sur tous les principaux systèmes d'exploitation (Linux, Windows, Mac, Android) et plateformes informatiques et peut servir pour des usages aussi divers que variés (cf paragraphe ci-dessous).

2.3 Exemples d'utilisation

Le langage Python est par exemple utilisé :

— lire, manipuler et administrer des fichiers (c'est l'utilisation de base de ce langage) ;
— pour le calcul scientifique et la manipulation et le traitement de données de grandes dimensions (la NASA utilise le langage Python, le Code Aster d'EDF est un logiciel libre de simulation numérique en mécanique des structures codé en Python, etc) ;
— dans des logiciels d'édition et/ou retouche d'image (GIMP, Inkscape ou Gedit sont codés en Python) ;
— dans des applications bureautiques (Dropbox est écrit en Python) ;
— dans des jeux vidéo (Civilization IV ou Battlefield 2 utilisent Python).
— pour le web (Google utilise Python , la plateforme YouTube est écrite en Python).
— pour les films d'animation (Pixar utilise Python pour produire ses films d'animation) ;
— par des constructeurs de composants informatiques (Intel, Hewlett-Packard, Seagate, IBM utilisent Python pour tester les performances de leurs nouveaux produits).

Pour d'autres exemples de logiciels utilisant Python comme langage de programmation, on pourra se reporter à la page Wikipedia ou à celle du Site officiel de Python.

2.4 Installation de Python

Lorsque Python est installé sur une machine, un interpréteur Python doit être aussi installé pour pouvoir exécuter du code écrit en Python. Un interpréteur est un programme qui lit le code et exécute les instructions qu'il contient. On peut le voir comme un programme qui exécute d'autres programmes.

L'installation de l'interpréteur Python dépend du système d'exploitation :

- Python est généralement installé de manière automatique sous Linux et Mac OS X ;
- sur Windows, l'installation de Python est généralement à faire par l'utilisateur.

Pour installer Python, rendez vous sur le site http://www.python.org pour télécharger la bonne version.

En dernier recours, si Python n'est pas déjà installé ou si vous n'arrivez vraiment pas à installer Python sur vos ordinateurs, vous pouvez l'utiliser directement en ligne, via les liens suivants www.online-python.com ou www.pythontutor.com.

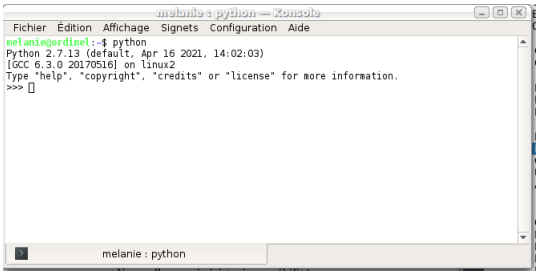
Attention il existe plusieurs versions de Python. La plus récente est celle numéro 3. Si vous utilisez une version plus ancienne, il se peut qu'il y ait des différences mineures qui apparaissent dans la syntaxe.

2.5 Écrire et exécuter un code Python

Nous allons voir ici trois possibilités.

2.5.1 L'invite de commande interactive

La première possibilité pour écrire et exécuter un code Python est d'utiliser l'invite de commande interactive. On y accède en tapant `python` dans une console.



Sous Windows on peut également y accéder en cliquant sur le menu démarrer puis Python.

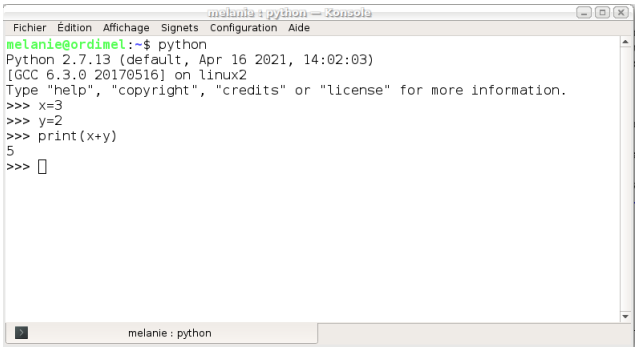
Un curseur apparaît juste après l'invite des commandes (**>>>**). C'est à cet endroit que vous pourrez lancer les instructions qui seront exécutées. Chaque commande doit se terminer par un retour à la ligne, ce qui nous limite à une commande par ligne (le retour à la ligne équivaut à demander à l'ordinateur d'exécuter la commande qui vient d'être tapée).

Ainsi les instructions sont interprétées et exécutées dès qu'elles ont été saisies.

Saisissons par exemple

```
>>> x = 3
>>> y = 2
>>> print (x + y)
5
```

Chaque ligne s'exécute.
Comme nous le verrons plus loin, ces lignes signifient la chose suivante : x reçoit la valeur 3, y la valeur 2 et on demande le calcul et l'affichage de $x + y$.
La ligne qui suit sans >>> renvoie le résultat de la dernière opération.



Pour quitter l'interpréteur Python interactif sur le terminal, on utilise Ctrl-D sur un poste Linux, ou Ctrl-Z + Entrer sur un poste Windows

Cette manière de faire est utile lorsque l'on souhaite tester des instructions à la volée, mais elle atteint vite ses limites, car elle en permet pas de sauvegarder le code et est peu efficace lorsqu'on souhaite écrire un programme en entier avec plusieurs lignes d'instructions (on est ici obligé de les écrire et de les exécuter une par une et si l'on se trompe sur une ligne de code, on a tout à refaire depuis le début).

2.5.2 Création d'un fichier .py puis exécution dans une console

Afin de palier aux problèmes précédemment évoqués, une solution consiste à utiliser un éditeur de texte (comme Edit sous Windows, Gedit sous Linux, TextEdit sous Mac, c'est à dire à écrire son programme dans un fichier à partir d'un éditeur de texte quelconque, de l'enregistrer au format *.py* (cela permet de sauvegarder son programme) et de l'exécuter dans une console en tapant la commande nom du *fichier.py* et en s'étant préalablement placé dans le répertoire contenant le fichier.

2.5.3 Utilisation d'un éditeur de programme Python

Des éditeurs de programme permettent à la fois d'éditer, d'exécuter et de débuger des programmes Python via une interface graphique agréable et pratique d'utilisation ont été développés.

Ces éditeurs se composent ainsi d'au moins deux fenêtres :

1. une appelée interpréteur qui joue le même rôle que la console et qui permet d'écrire de petites instructions, de faire des calculs, etc ;
2. une appelée éditeur, qui joue le rôle de l'éditeur de texte et qui permet d'écrire des programmes sous format texte, de les enregistrer, de les modifier sans problème et de les exécuter.

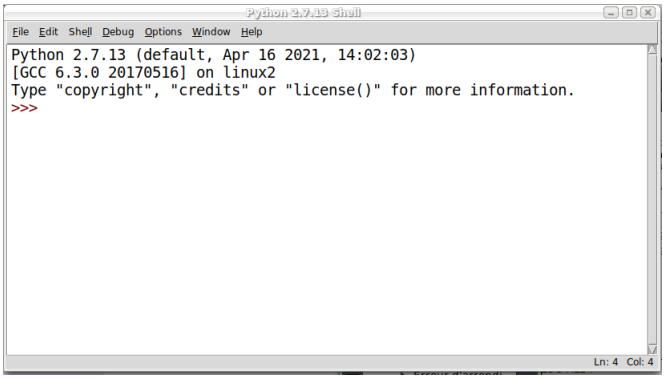
A noter que lors de l'exécution d'un programme saisie dans la fenêtre éditeur, les messages d'erreurs ou d'informations s'affichent dans la console.
Suivant les éditeurs, d'autres fenêtres peuvent s'y ajouter : arborescence des fichiers, historiques des commandes, etc.

Le plus connu et qui est inclus à l'installation standard de Python sur la plupart des distributions (Linux, Windows, Mac) est IDLE.

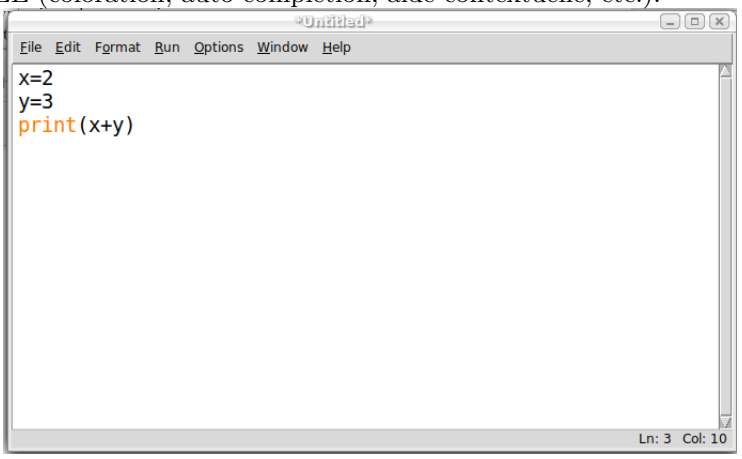
En faisant une recherche des logiciels installés sur votre ordinateur vous devriez le trouver sans problème, autrement n'hésitez pas à l'installer.

A l'ouverture, on retrouve une console Python, semblable à celle de l'invite de commande interactive, avec cependant les différences notables suivantes :

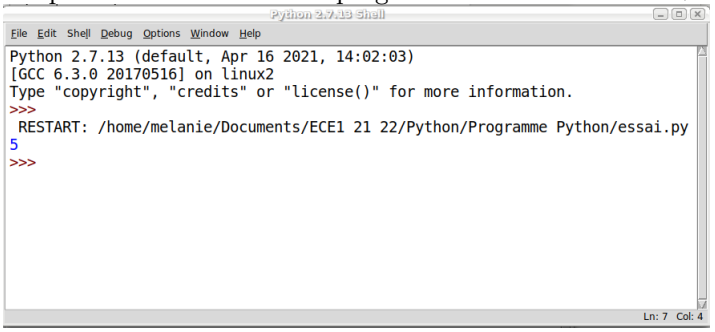
- coloration syntaxique des expressions ;
- auto-complétion avec la touche *Tab* ;
- aide contextuelle lors de l'ouverture d'une parenthèse ;
- liste de choix lorsque l'on veut utiliser une méthode ou un attribut.
- bibliothèques déjà installées pour la version de Python utilisée faciles à parcourir.



Pour créer un programme Python, il faut actionner le menu *File > New File* qui ouvre une seconde fenêtre fonctionnant comme un éditeur de texte avec les même avantages que la console IDLE (coloration, auto-complétion, aide contextuelle, etc.).



On peut ensuite exécuter le programme via le menu *Run > Run Module*.



L'interface et le fonctionnement d'IDLE peuvent être légèrement personnalisés pour rendre son fonctionnement plus agréable via le menu *Options > Configure IDLE*.

Deux autres éditeurs parmi les plus utilisés sont Spyder (https://www.spyder-ide.org/) et Pyzo (https://pyzo.org/) . Mais il en existe de nombreux autres (PyScripter, PyCharm, PyDev pour ne citer qu'eux). A chacun de choisir celui qui lui semble le plus maniable.

2.5.4 Au lycée

Au lycée, nous serons sur des postes Windows avec Python et un logiciel de programmation déjà installé.

2.6 Aide Python

Deux possibilités pour rechercher de l'aide pour Python :

1. Vous trouverez une aide complète à l'adresse suivante `https://docs.python.org/`. Vous pouvez rentrer sur cette page des mots-clés (nom de fonctions, modules, etc) afin d'en trouver l'aide associer.
2. Lorsque vous êtes sur la console ou sous un éditeur de programme, vous pouvez utiliser la commande `help()` qui affiche l'aide spécifique à une fonction ou à l'ensemble des fonctions d'un module.
   - `help(print)` aide sur une fonction
   - `help(math)` aide sur le module

Chapitre 3

Variables, type de données et structure du code en Python

3.1 Variables et affectations

Nous allons aborder ici la notion de variables.

3.1.1 Définition

Les données qu'utilise un programme peuvent varier au cours de son exécution. On les appelle alors des variables. Plus précisément en informatique, Une variable est un nom associé à un emplacement de la mémoire. C'est comme une boîte que l'on identifie par une étiquette.

Pour affecter une valeur à une variable, on utilise le signe **=** :

```
>>> x = 2
```

Ainsi a commande $x = 2$ signifie que j'ai une variable x associée à la valeur 2.
Il est important de faire la distinction entre les variables et les objets liés : dans $x = 2$, la variable x ne contient pas directement la valeur2, mais un objet 2 est associé à cette variable.
Plus précisément, lorsque l'instruction $x = 2$ est saisie, les étapes suivantes sont enchaînées :

- création d'un objet représentant la valeur 2 ;
- création d'une variable x, si elle n'existe pas déjà ;
- association de la variable x à l'objet 2.

Ainsi, les variables sont des symboles qui associent un nom (afin que le programmateur sache quelle donnée il manipule) à un objet (qui lui sera manipulé par l'ordinateur).

3.1.2 Manipulation des variables et affectation

On peut créer plusieurs affectations dans la même instruction en séparant les noms des variables et les valeurs par des virgules :

```
>>> a ,b =2 ,3
>>> a
2
>>> b
3
```

équivaut à **a = 3** et **b = 2**.

Ce mécanisme peut paraître anodin, mais il permet de réaliser très simplement une permutation de valeurs sans avoir recours a une variable intermédiaire :

```
>>> a = 2
>>> b = 3
>>> a ,b=b ,a
>>> a
3
>>> b
```

```
2
```

On peut affecter le même objet dans plusieurs variables, en utilisant la syntaxe suivante :

```
>>> a = b = c = 3
```

qui équivaut donc à :

```
>>> c = 3
>>> b = c
>>> a = b
```

3.1.3 Nom d'une variable

Voici les règles que doit vérifier le nom d'une variable en Python :
- ne peut contenir que des lettres, des chiffres et des blancs soulignés ;
- ne peut pas commencer par un chiffre ;
- ne peut pas contenir d'espace, d'accent ou de caractères spéciaux ;
- distingue minuscules et majuscules ;

```
>>> a =3
>>> a
3
>>> A
NameError : name 'A ' is not defined
```

En maths, on donne généralement seulement une lettre à une inconnue (x par exemple). Dans un programme il est préférable d'utiliser des noms pour une meilleure lisibilité (par exemple liste pour une liste, moyenne pour une moyenne, etc).

3.1.4 Noms de variables réservés à Python

Certains noms sont réservés à Python et il n'est alors pas possible de créer une variable portant l'un de ces noms.

Voici la liste de ces noms.

and	else	in	return
as	except	is	true
assert	false	lambda	try
break	nally	none	while
class	for	nonlocal	with
continue	from	not	yield
def	global	or	
del	if	pass	
elif	import	raise	

Table 3.1 Noms réservés

3.1.5 La commande print

Lorsque l'on est directement sur la console, on peut faire afficher le contenu de la variable en tapant simplement son nom en ligne de commande.

```
>>> a =3
>>> a
3
```

Par contre si l'on tape ces commandes dans un fichier texte

```
a =3
a
```

et qu'on l'exécute dans la console, alors la valeur de **a** ne s'affichera pas.

Voici en effet ce que vous obtiendrez dans la console

```
>>>
```

Pour la voir apparaître, il faut utiliser la commande print et ajouter la ligne suivante dans le fichier texte :

```
 a =3
print ( a)
```

Voici alors ce qui s'affichera dans la console

```
>>> a =3
>>> print (a)
3
```

Cela est vrai pour les variables, mais aussi pour les calculs directs (on écrira `print(2*3)` et non pas seulement `2*3`.

Si l'on souhaite une lecture plus précise du résultat, on peut taper

```
a =3
print ( 'a= ', a)
```

qui affiche la chaîne de caractère `a=` (cf. Chapitre 6) puis la valeur de la variable `a` , c'est à dire ici 3.

De manière plus générale, la commande **print** permet d'afficher du texte à l'écran et peut prendre plusieurs paramètres séparés par une virgule qui seront concaténés.

```
>>> print ( 'Bonjour ','le  monde ')
Bonjour le monde
```

3.1.6 La commande input

Il se peut que dans un programme on souhaite que l'objet associé à une variable soit fourni par l'utilisateur. Dans ce cas on utilise la commande **input** dont voici un exemple de syntaxe

```
n= input ( ' entrer un  entier ')
```

Lorsqu'on exécute cette commande le message "entrer un entier" s'affiche dans la console, l'utilisateur rentre alors la valeur de l'entier souhaité, qui est alors affecté à la variable n.

Ainsi la commande `input` permet d'afficher un message à l'écran et de donner la main à l'utilisateur pour qu'il saisisse des caractères au clavier et qu'il appuie sur la touche entrée pour affecter la valeur choisie à la variable.

```
>>> c= input ( ' Veuillez saisir  un  entier : ')
Veuillez  saisir  un  entier :2
>>> c
'2 '
>>> c= input ( ' Veuillez saisir  une  chaine :  ')
Veuillez  saisir  une  chaine :  coucou
>>> c
' coucou '
```

3.2 La commande type

3.2.1 Définition

Une variable peut contenir des objets très différents : un réel, un entier, une matrice, une chaîne de caractère, un booléen (vraie ou faux), etc.

Afin de distinguer et de spécifier les différents contenus (entiers, réels, booléens, chaines de caractères, etc), toute variable utilisée par le programme doit posséder un type.

Certains langages de programmation nécessitent que l'on spécifie le type. Avec Python, il n'est pas nécessaire de déclarer le type d'une variable car l'interpréteur Python détecte lui-même le type de la variable utilisée.

Ainsi, lorsque l'on écrit `x = 2` l'interpréteur Python détecte que `2` est un entier et associe donc ce type à la variable x.

Pour voir le type d'une donnée, on utilise la commande `type`

```
>>> a = 2
>>> b = 2.0
>>> type (a)
< class ' int '>
>>> type (b)
< class ' float '>
```

On voit bien ainsi que Python fait une distinction entre les **nombres entiers** (`int`) et les **nombres réels** (`float`).

3.2.2 Principaux types

Type	Exemples de valeur
Nombres (entiers, réels, etc)	`1, 1.4523, -2.5`
Chaîne de caractère	`'bonjour', 'phrase'`
Liste	`[1, 2, 3], ['a', 'b'],`
Tuple	`(1, 2), ('a', 'b', 'c')`
Ensemble	`set('abc'), 'a', 'b', 'c'`
Dictionnaire	`1 :'a', 2 :'b'`
Fichier	`open('fichier.txt')`
Booléen	`True, False`
Rien	`None`

Table 3.2 différents types de données

3.2.3 Le type None

Python propose un type spécial qui ne peut prendre qu'une seule valeur possible `None` (qui correspond à la valeur rien). Ce type de variable est renvoyé par défaut lorsqu'une opération n'a pas pu être menée à son terme pour dire que la variable ne contient rien. L'usage principal est ainsi de l'utiliser pour vérifier que des parties de programme ont correctement fonctionné.

3.3 Structure du code et commentaires

3.3.1 Structure

Un programme consiste en une suite d'instructions.

En Python, il n'y a pas de symbole de fin d'instructions, c'est le passage à la ligne suivante qui joue ce rôle.

```
instruction1
instruction2
instruction3
```

L'indentation elle est utilisée pour définir un bloc de code : instructions avec conditions (`if`), répétitions un nombre donnée de fois d'une suite d'instructions (`for`), répétitions d'une suite d'instructions tant qu'une condition n'est pas vérifiée (`while`), etc.
De plus la ligne d'en tête du bloc se termine par deux points (:).

```
ligne_d_en_tete :
    bloc_d_instructions
```

La fin de l'indentation correspond à la fin du bloc d'instructions. Il n'y a besoin de rien de plus (notamment pas de end comme dans certains langages).

```
if  condition :
    instruction1
    instruction2
```

Ainsi la structure d'un code en Python est simple, homogène et permet de rendre le programme plus lisible.

De plus, l'éditeur fera automatiquement l'indentation après que vous ayez tapés les `:` .
Il vous faudra juste penser à revenir bien à gauche pour les instructions qui suivent les dernières du bloc.

Les espaces et sauts de ligne sont ignorés par l'interpréteur Python.

```
ligne_d_en_tete :
    instruction1
instruction2
```

et

```
ligne_d_en_tete :
    instruction1

instruction2
```

sont équivalents à

```
ligne_d_en_tete :
  instruction1
  instruction2
```

présentation qu'il faut privilégier pour une meilleure lisibilité.

3.3.2 Commentaires

Enfin, Python permet d'intégrer des commentaires au sein du code, qui seront ignorés par l'interpréteur. Cela permet d'augmenter la lisibilité d'un programme par une tierce personne ou par soi même (si l'on doit le réutiliser longtemps après l'avoir écrit) en expliquant les différentes instructions ou ce que représentent chacune des variables par exemple.

Pour cela il faut débuter le commentaire pas un caractère `#`. Et ce dernier se terminera tout simple à la fin de la ligne.

```
ligne_d_en_tete :
  instruction1     # première  instruction
  instruction2
```

Chapitre 4

Les principaux types numériques et booléens et opérations de base associées

Ce chapitre va nous permettre de commencer à utiliser le langage Python. Nous nous limiterons pour commencer à des opérations basiques comme des opérations mathématiques.

4.1 Les types numériques Réels et Entiers

Parmi les nombres, Python définit des types plus précis répertoriés dans le tableau ci-dessous :

Type	**Exemple**
Integer	`1, -2, 568742`
Float	`1.0, 3.12, -5.6, 15522.17885`
Complex	`2 + j`
Decimal	`Decimal('0.1')`
Fraction	`Fraction(1, 2)`

Table 4.1 Principaux types de nombres

Python fait une distinction entre les **nombres entiers** (**`int`**) et les **nombres réels** (**`float`**).

```
>>> a = 2
>>> b = 2.0
>>> type (a)
< class ' int '>
>>> type (b)
< class ' float '>
```

Les fonctions **`int()`** et **`float()`** permettent elles de convertir un nombre quelconque en entier ou réel respectivement.

```
>>> a = 2
>>> type (a)
< class ' int '>
>>> b= float (2)
>>> type (b)
< class ' float '>
```

4.2 Opérations

4.2.1 Opérations classiques

Les valeurs de ces variables peuvent être modifiées au moyen d'une série d'opérations.
Les commandes Python associées aux opérations mathématiques classiques sont les suivantes :

1. **+** pour l'addition
2. **-** pour la soustraction
3. ***** pour la multiplication
4. $**$ pour la puissance
5. **/** pour la division
6. **//** pour le quotient de la division euclidienne
7. % pour le reste de la division euclidienne

4.2.2 Opérations augmentées

Python offre également la possibilité d'effectuer des **affectations augmentées** : Les deux instructions de l'exemple suivant sont équivalentes :

```
>>> a = 2
>>> a = a + 3
>>> a
5
```

```
>>> a = 2
>>> a += 3     # affectation   augmentée
5
```

Les affectations augmentées fonctionnent avec tous les types de données et tous les opérateurs standards : **+= -= *= /= //= %=** etc.

4.2.3 Constantes et puissances

Les constantes usuelles π, e, etc ne sont pas directement accessibles. Il faut pour les manipuler importer la bibliothèque Numpy (cf. Chapitre 9 et Chapitre 10) pour la notion de bibliothèque et pour Numpy en particulier).

On utilisera la lettre e pour signaler un exposant en base 10. Ainsi **2e1** signifie $2*10^1$ et représente donc 20.

```
>>> 2 e1
20.0
```

4.3 Les types numériques Décimal et Fractions

4.3.1 Décimal

Python possède également un type **decimal** qui permet de gérer des nombres décimaux avec une précision définie par l'utilisateur.

Cela est particulièrement utile dans les cas où la conversion des nombres en binaires par la machine introduit des erreurs :

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783 e -17
```

Cela est du au fait que la valeur réellement stockée par l'ordinateur derrière 0.1 n'est pas exactement 0.1 (les flottants sont arrondis en langage machine). Pour le voir il faut importer la fonction **Decimal** du module Decimal (cf. Chapitre 9 pour la notion de module) :

```
>>> from decimal import Decimal
>>> Decimal (0.1)
Decimal ( ' 0.1000000000000000055511151231257827021181583404541015625 ')
```

Pour manipuler le nombre décimal 0.1, il faut utiliser cette même fonction Decimal mais avec des apostrophes :

```
>>> Decimal ( ' 0.1 ')+ Decimal ( ' 0.1 ')+ Decimal ( ' 0.1 ') - Decimal ( ' 0.3 ')
Decimal ( ' 0.0 ')
```

Si l'on ne souhaite juste afficher 0.0 et non Decimal('0.0'), on peut utiliser la commande **print**:

```
>>> print ( Decimal ( ' 0.1 ')+ Decimal ( ' 0.1 ')+ Decimal ( ' 0.1 ') - Decimal ( '
    0.3 '))
0.0
```

4.3.2 Fractions

De même que pour la manipulation des décimaux, la manipulation des fractions sans perte de précision (c'est à dire en gardant l'écriture fractionnaire) nécessite d'utiliser la fonction **Fraction** du module Fraction :

```
>>> from fractions import Fraction
>>> Fraction (1 , 3) + Fraction (1 , 2)
Fraction (5 , 6)
>>> print ( Fraction (1 , 3) + Fraction (1 , 2) )
5/6
```

Le résultat renvoyé est la fraction 5/6 que l'on peut comparer pour vérifier avec l'écriture décimale en utilisant la commande oat :

```
>>>  1 ./3 + 1 ./2
0.8333333333333333
>>> float ( Fraction (1 , 3) + Fraction (1 , 2) )
0.8333333333333333
```

4.4 Le type booléen

Le terme booléen s'applique à un langage informatique binaire, inventé par le mathématicien George Boole, qui consiste à programmer des variables qui ne peuvent prendre que deux états (généralement notés vrai et faux), destinés à représenter les valeurs de vérité de la logique.

En Python ces deux états sont **True** et **False**.

```
>>> type ( True )
< class ' bool '>
```

En Python, les booléens **True** et **False** ont le même comportement que les entiers 1 et 0.

```
>>> float ( True )
1.0
>>> float ( False )
0.0
>>> int ( True )
1
>>> int ( False )
0
```

Et peuvent donc être manipuler comme tel.

```
>>> False + 5
5
>>> True *2
2
```

4.5 Opérateurs logiques et de comparaison

Opérateur	Signification
>	Supérieur
<	Inférieur
>=	Supérieur ou égal
<=	Inférieur ou égal
==	Egal
!= ou <>	différent de
is	Objet identique
is not	Objet différent
A and B	vrai si A et B sont vrai
A or B	est vrai si A ou B est vrai
not A	vrai si A est faux

Table 4.2 Quelques opérateurs classiques

A noter que Python comprend les conditions sous forme de double encadrement.

```
>>> (3 <1)
False
>>> (4==2*2)
True
>>> not 3==2
True
>>> (4==2*2) and ( not 3==2)
True
>>> (3 <1) or (4==2*2)
True
>>> (3 <1) or (4==5)
False
```

Ne pas confondre **=** (affectation) avec == (test)

```
>>> 4=2
SyntaxError : cannot assign to literal
>>> 4==2
False
```

Ainsi on ne confondra pas **x= 2** (affectation de la valeur 2 à **x**) et **x==2** (condition vraie si **x** égale 2).

Chapitre 5

Les listes et les ranges

Nous allons ici évoquer la notion de listes (**list**) et de range (**range**), ainsi que les opérations que l'on peut faire dessus.

5.1 Définition d'une liste

Une liste est définie par un ensemble de valeurs éventuellement de types différents (nombres, chaines de caractère, etc) entre deux crochets :

```
>>> liste = [" chat " ,1, " chien " ,2]
>>> type ( liste )
< class ' list '>
>>> liste
[ 'chat , 1, ' chien ', 2]
```

Les listes peuvent être imbriquées :

```
>>> liste =[1 , 2, 3]
>>> m = [ liste , [3 , 4, 5, 6]]
>>> m
[[1 , 2 , 3] , [3 , 4 , 5 , 6]]
```

Pour créer une liste vide, on utilise des crochets sans rien à l'intérieur :

```
>>> l = []
>>> type (l)
< class ' list '>
```

5.2 Propriétés des listes

Une liste en Python est un élément qui est :
- **itérable**, car elle est capable de retourner les éléments qui la composent les uns à la suite des autres ;
- **indexable**, car elle est capable de retourner n'importe lequel de ses éléments sans avoir à parcourir l'ensemble de la séquence ;
- **modifiable**, car les éléments d'une liste peuvent être modifiés sans avoir à être redéfinis (on peut ajouter, supprimer des éléments).

On accède à un élément de la liste à l'aide des crochets **[ ]**.
A noter que l'indexation d'une liste commence à 0 et que les indices des listes doivent être des entiers

```
>>> l = [1 ,2 ,3]
>>> l [0]
1
>>> l [1]
2
>>> l [2]
```

```
3
>>> l [2.0]
TypeError : list indices must be integers , not float
>>> l [4]
IndexError : list index out of range
>>> l [2]=4
>>> l
[1 , 2, 4]
```

5.3 Opérations sur les listes

La table 6.1 référencent les principales opérations que l'on peut faire sur une liste :

Opérateur	Signification
`l + m`	concaténation de l et m
`l * m`	ajouter m fois l à elle-même
`l[i]`	élément à la i-ème position
`l[i]=x`	remplace l'élément à la i-ème position par l'élément x
`del(l[i])`	supprime l'élément à la i-ème position de la liste l
`l[i :j]`	sous-séquence de l, de la i-ème à la j-1-ème position
`l[i :j]=t`	remplace la sous-séquence de l, de la i-ème à la j-1-ème position par la liste t
`del(l[i :j])`	supprime les éléments de la liste l qui sont de la i-ème à la j-1-ème position
`l[i :j :k]`	sous-séquence de l, de la i-ème à la j-1-ème position avec un pas de k
`l[i :]`	sous-séquence de l de la i-ème position jusqu'à la dernière
`len(l)`	longueur = nombre d'éléments de la séquence l
`l.append(x)`	Ajoute l'élément x à la fin de la liste l

Table 5.1 Opérations classiques sur les listes

Attention, quand on parle de la ième position, comme les indices commence à 0, il s'agit en fait de la i+1 ème position dans notre façon de compter classique en démarrant à 1.

Autres opérateurs :

Opérateur	Signification
`x in l`	x est dans l
`x not in l`	x n'est pas dans l
`min(l)`	plus petit élément de la séquence l
`max(l)`	plus grand élément de la séquence l
`sum(l)`	somme des éléments de la liste l
`l.index(x, i, j)`	index de la première occurrence de x dans l, à partir de l'indice i et jusqu'à j
`l.count(x)`	nombre d'occurrence de x dans l
`l.extend(t) .`	Ajoute les éléments de la liste t en fin de liste l (commande redondante avec +)
`l.insert(i, x)`	Insertion de l'élément x à la position i
`l.remove(x)`	Suppression de tous les éléments x
`l.pop(i)`	Retourne et supprime de la liste l'élément à la position i
`l.clear()`	Vide la liste
`l.reverse()`	Inverse l'ordre des éléments de la liste
`l.copy()`	Crée une copie de la liste
`l.sort()`	Trie les éléments de la liste par ordre croissant

Table 5.2 Autres opérations

5.4 Les ranges

Les **range** sont des séquences de nombres entiers, définies à l'aide d'une borne initiale, d'une borne finale et d'un pas. Si la borne initiale est omise, la valeur par défaut sera zéro. Le pas par défaut vaut quant à lui 1. A minima il faut ainsi toujours donner la borne finale.

```
>>> r = range (5)
>>> type (r)
< class ' range '>
```

On voit bien ici que les ranges ne sont pas du même type que les listes.

Pour visualiser le contenu d'un objet de type **range**, on le convertit en liste à l'aide de la fonction **list()**:

```
>>> a = list ( range (5) )
>>> a
[0 , 1, 2, 3, 4]
>>> b = list ( range (4 ,6) )
>>> b
[4 , 5]
>>> c = list ( range (2 ,10 , 2) )
>>> c
[2 , 4, 6, 8]
```

On notera en particulier que la borne d'arrivée n'est pas atteinte : la commande **range(a,b,c)** renvoie la séquence des entiers compris entre **a** et **b-1** (attention pas **b**) avec un pas de c

Les **range** sont couramment utilisés comme compteur dans les boucles **for** (cf. Chapitre 7).

Chapitre 6

Les chaines de caractères

Nous allons maintenant étudier un autre type de données fondamentales : les chaines de caractères (type `str`).

6.1 Définition

Les apostrophes (`'...'`) et guillemets (`"..."`) s'utilisent indifféremment pour définir une chaîne de caractères :

```
>>> " bonjour le monde "
' bonjour le monde '
>>> ' bonjour le monde '
' bonjour le monde '
```

```
>>> " bonjour le monde " == ' bonjour le monde '
True
```

L'utilisation conjointe des apostrophes et des guillemets permet d'écrire des chaînes de caractères qui contiennent elles mêmes des guillemets ou des apostrophes.

```
>>> 'l'equation '
SyntaxError : invalid syntax
>>> "l'equation "
"l'equation "
>>> " elle a dit : " demain il verra beau ""
SyntaxError : invalid syntax
>>> ' elle a dit : " demain il verra beau " '
' elle a dit : " demain il verra beau " '
```

On peut aussi utiliser `\'` pour distinguer l'apostrophe comme chaine de caractère et l'apostrophe comme signal de début ou de fin de chaines de caractère.

```
>>> 'l\' equation '
"l'equation "
```

Pour ne pas faire apparaître les apostrophes ou les guillemets de début et fin de chaine de caractère, utilisez la commande **`print`**

```
>>> ' bonjour le monde '
' bonjour le monde '
>>> print ( ' bonjour le monde ')
bonjour le monde
```

On peut tout à fait affecter une chaine de caractère à une variable.

```
>>> chaine = ' bonjour le monde '
>>> print ( chaine )
bonjour le monde
```

6.2 Opérations sur les chaines de caractères

6.2.1 Opérations de base

On peut effectuer de nombreuses opérations sur les chaines de caractère, comme regarder leur longueur, tester la présence d'un caractère ou d'un groupe de caractères dans une chaîne, accéder au caractère situé à une position donnée dans la chaîne, etc.
Voici une liste non exhaustive des principaux opérateurs applicables aux chaînes de caractères

Opérateur	**Signification**
`l+m`	Concaténation de chaînes de caractères
`l*n`	Répétition n fois d'une chaîne de caractères l
`m in l`	Inclusion d'une chaîne m dans une chaine l
`m not in l`	Non inclusion d'une chaîne m dans une chaine l
`m[i]`	Caractère à la i-ième position dans la chaîne m
`m[i : j]`	Caractères compris entre les i-ème et j-1-ème positions de la liste m
`len(m)`	Donne la longueur de la chaine m

Table 6.1 Opérations classiques

Attention, comme pour les listes les indices débutent à 0. De ce fait, lorsque l'on parle de la ième position, il s'agit en fait de la i+1 ème position dans notre façon de compter classique en démarrant à 1.

Voici des exemples d'utilisation de ces opérations :

```
>>> 'le resultat vaut '+ '. '
'le resultat vaut . '
>>> 'le resultat '*2
'le resultat le resultat '
>>> 're ' in ' resultat '
True
>>> ' nom 'not in ' resultat '
True
>>> ' resultat ' [3]
'u '
```

Autre exemple :

```
>>> a=" Hello "
>>> b= ' World ! '
>>> a+b
' Hello World ! '
>>> 3* a
' Hello Hello Hello '
>>> a [0]
H
>>> b [2]
r
>>> len (b)
7
```

On notera que les opérations sur les chaines de caractère sont similaires à celles sur les listes. Une chaine de caractère peut être vue comme une liste non modifiable. Les opérations similaires à celle des listes s'appliquent, mais on ne peut pas modifier un élément de la chaine comme pour une liste par une simple affectation.

```
>>> chaine = ' liste '
>>> chaine [1]
'i '
>>> chaine [1]=2

Traceback ( most recent call last ):
  File " < pyshell #6 > "line 1, in < module >
    chaine [1]=2
TypeError : ' str 'object does not support item assignment
```

6.2.2 Fonctions appliquées à une chaine de caractère

On peut appliquer de multiples fonctions à une chaine de caractère afin de compter le nombre d'occurrences d'une lettre dans la chaine, remplacer un caractère par un autre, etc.

Voici une liste non exhaustive des principales fonctions que l'on peut appliquer à une chaine de caractère

Fonction	Signification
`chaine.count(s)`	Compte le nombre d'occurrence du caractère s dans la chaîne
`chaine.nd(s)`	Retourne l'indice de la première occurrence du caractère s dans la chaîne ou -1 si elle n'est pas présente
`chaine.isalpha()`	Retourne True si la chaîne est composée exclusivement de lettres
`chaine.isdigit()`	Retourne True si la chaîne est composée exclusivement de chiffres
`chaine.replace(a, b)`	Retourne un copie de la chaîne où les caractères a ont été remplacés par b
`chaine.upper()`	Remplace les minuscules par des majuscules
`chaine.lower()`	Remplace les majuscules par des minuscules
`chaine.capitalize()`	Met en majuscule la première lettre
`chaine.endwith(s)`	Retourne vrai si la chaîne se termine par les caractères s
`chaine.startwith(s)`	Retourne vrai si la chaîne commence par les caractères s

Table 6.2 Quelques exemples de fonctions

```
>>> chaine = ' Que de fonctions dans Python ! '
>>> print ( chaine )
Que de fonctions dans Python !
>>> chaine . count ( 'e ')
2
>>> chaine . find ( 's ')
15
>>> chaine . upper ()
' QUE DE FONCTIONS DANS PYTHON ! '
>>> chaine . lower ()
' que de fonctions dans python ! '
>>> chaine . replace ( 'e ','a ')
' Qua da fonctions dans Python ! '
```

6.3 Convertir une variable en chaîne de caractère

Il est possible de convertir le contenu d'une variable quelconque en chaîne de caractère en utilisant la fonction **`str()`**.

```
>>> a = 4.16
>>> b = str (a)
>>> type (b)
< class ' str '>
```

```
>>> a =2
>>> chaine = 'le resultat est : '
>>> chaine +a
Traceback ( most recent call last ):
  File " < pyshell #6 > "line 1, in < module >
    chaine +a
TypeError : cannot concatenate ' str ' and ' int ' objects
>>> chaine + str (a)
'le resultat est :2 '
```

6.4 Les anti-slashs

Afin de signaler un caractère spécial, on utilise les anti-slashs :

\n pour une nouvelle ligne ;
\t pour une tabulation ;
**** pour un anti-slash ;
\' pour une apostrophe
etc.

Cela peut entraîner des erreurs lorsque l'on a à manipuler une chaine de caractère comportant déjà des antislashs qui doivent être compris comme tel (ce qui est souvent le cas avec le nom du chemin d'un fichier par exemple).

Dans ce cas, afin que Python interprète les **** comme de simples ****, il faut faire apparaître un `r'` juste devant la chaine de caractère :

```
>>> print ( 'C :\theatre\nina ')
C:   heatre
ina
>>> print (r 'C :\theatre\nina ')
C:\theatre\nina
```

Chapitre 7

Syntaxe Python

7.1 Les tests

Les tests permettent de mettre des conditions à l'exécution d'instructions. La syntaxe Python prend la forme suivante :

```
if  condition :
     instruction1
     instruction2
     ...
```

Ainsi la condition est vérifiée alors les instructions 1 et 2 seront exécutées.

Si des instructions sont à exécuter lorsque la condition n'est pas réalisée, on ajoute la syntaxe **else** :

```
>>>  x  =  1
>>>  if  x  >=  0:
     print ( "x positif ")
     else :
     print ( "x négatif ")

x  positif
```

A noter que le **else** n'est pas suivi d'une condition, car le sinon revient à dire que si la condition première est fausse (autrement dit si la condition inverse de la première est vérifiée) alors on applique les instructions qui suivent.

Si l'on doit enchaîner plusieurs tests, on ajoute **elif** :

```
x =5
y =2
if  x == y:
    print ( "x é gale y")
elif  x > y :
    print ( "x est  strictement   plus  grand  que  y")
else :
    print ( "x est  strictement   plus  petit  que  y")

x  est  strictement   plus  grand  que  y
```

Le décalage des instructions à exécuter dans **if**, **elif** et **else** est indispensable pour obtenir un résultat correct.

Lorsque aucune instruction ne doit être exécutée quand une condition est réalisée, on utilise pass et c'est obligatoire de l'écrire, sinon si l'on ne met rien l'ordinateur affichera une erreur.

```
if  x  >  0:
     print ( " signe positif " )
elif  x ==0:
     pass
else :
     print ( " signe né gatif " )
```

Notons également que la présence des parenthèses autour de la condition est optionnel en Python. Ainsi les deux instructions suivantes seront identiques :

```
if x > 3:
    ...
if (x > 3) :
    ...
```

7.2 Les boucles

Les boucles permettent d'exécuter plusieurs fois un même bloc d'instructions.

7.2.1 La boucle for

La boucle **for** permet d'exécuter un même bloc d'instructions pour tous les éléments d'un objet itérable (liste, range, chaîne, etc).
Sa syntaxe est la suivante :

```
for element in iterable :
    bloc_instructions
```

Un usage fréquent des boucles `for` est le parcours de séquences d'entiers (lorsqu'on veut par exemple répéter les instructions un nombre n donnée de fois). On utilise alors un **range** :

```
>>> for i in range (0 , 10 , 2) :
...     print (i)
... else :
...     print ( ' Fin ')
...
0
2
4
6
8
Fin
```

Ainsi les instructions sont effectuées pour la première fois pour i=0, puis i=2, etc (car **range(0, 10, 2)** comprend la séquence 0,2,4,6,8,10).
A noter que les instructions peuvent dépendre de i ou non. On préférera donner un nom à la séquence itérable (par exemple S) et écrire :

```
S= range (0 ,10 ,2)
for i in S
...
```

Si l'on souhaite répéter n fois une suite d'opérations, on utilisera **S=range(1,n)**.
Mais on peut aussi utiliser les boucles **for** avec une chaîne de caractères :

```
>>> mot = ' ECG1 '
>>> for l in mot :
    print (l)

E
C
G
1
```

Ou avec des listes

```
>>> liste = [" Un " ," Deux " ," Trois "]
>>> for l in liste :
        print (l)

Un
Deux
Trois
```

7.2.2 La boucle while

La boucle `while` permet d'exécuter un bloc d'instructions tant qu'une condition est réalisée (on ne sait alors pas a priori combien de fois il faudra le répéter). Sa syntaxe est la suivante :

```
while  condition :
    instructions
```

Cela donne par exemple :

```
>>> a = 0
>>> while  a < 8:
    print ( a)
    a=a + 2

0
2
4
6
```

A noter que :

Le bloc d'instructions qui suit la boucle while peut ne pas être exécutée si la condition de départ n'est pas vérifiée dès le début.

Avant d'exécuter une boucle while, il faut vérifier que la condition devient fausse au bout d'un certain temps, sinon le programme s'exécutera indéniment.

Pour connaître le nombre de fois où la boucle va s'exécuter, il suffit de mettre un compteur, qui est initialisé avant le while, (c=0), puis incrémenté à chaque passage dans la boucle (**c=c+1**).

7.3 Commandes break, else et continue

Afin de sortir prématurément d'une boucle (for ou while), on peut utiliser la commande **break:** Par exemple, si la syntaxe **if a==0:break** apparaît dans la boucle while, lorsque la variable a vaut 0, la boucle est interrompue.

On peut utiliser la commande **continue:**pour imposer le retour au début des instructions constituant la boucle.
Par exemple,si la syntaxe **if a==1:continue** apparaît dans la boucle while, lorsque la variable a vaut 1, la boucle passe à l'étape suivante, sans avoir exécuté la boucle pour $a = 1$.

Voici un exemple d'utilisation :

```
>>> for  i  in  range (0 ,10) :
...       if  i ==2:
...            continue
...       if  i ==4:
...            break
...       print (i)
...
0
1
3
```

L'ajoute de la commande **else** dans un boucle permet de définir un bloc d'instructions qui sera exécuté à la fin seulement si la boucle s'est déroulée complétement sans être interrompue par un **break**

Il est important de noter dans ce cas la différence entre l'ajout du **else** dans la boulce et le fait d'ajouter des instructions à exécuter à la toute fin de la boucle. En effet, les instructions présentes après la boucle, s'exécutent dans tous les cas (avec ou sans interruption par un **break**). Par contre, le bloc d'instructions défini après le **else** ne s'exécutera pas lors de l'interruption par un **break**

Par exemple, imaginons que l'on recherche dans une liste un élément particulier, par exemple le mot "ECG"

```
>>> for mot in [" MPSI " ," BCPST " ," ECG " ," PCSI " ," TB " ]:
      if mot == " ECG ":
            print (" ECG trouvé !" )
            break
      else :
            print (" ECG introuvable ")
...
ECG trouvé !
```

Ce code n'affichera "ECG introuvable" que si l'intégralité de la boucle a été parcourue sans trouver l'élément recherché dans la liste. Si la valeur a été trouvé, le programme affichera "ECG trouvé !". boucle.

Chapitre 8

Fonctions

Nous allons dans ce chapitre définir ce qu'est une fonction et apprendre à en créer.

8.1 Définition, syntaxe

8.1.1 Définition

Une fonction est un groupe d'instructions qui peut être exécuté plusieurs fois dans un programme. Plus précisément, une fonction est un objet qui permet de spécifier des paramètres en entrée, d'y appliquer une succession d'instruction et donner un résultat en sortie (qui pourra ainsi varier d'une exécution à l'autre en fonction des paramètres en entrée).

Nous en avons déjà manipulé sans le savoir. En effet, la commande **print()** qui permet d'afficher le contenu d'une variable donnée en entrée ou celle **type()** qui permet de connaître le type d'une donnée données en entrée ou celle **input()** sont des fonctions qui ont été prédéfinies dans Python.

Une fonction en Python débute par la mot clé **def** suivi du nom de la fonction et de parenthèses. Les instructions qui forment le corps de la fonction (c'est à dire celle qui seront exécutées lorsque la fonction sera appelée dans un programme) débutent à la ligne suivante et se termine lorsque l'indentation retourne à son niveau initial :

```
def  ma_fonction () :
     instruction_1
     instruction_2
     ...
     instruction_n
```

La convention, qui est la même que pour le nom des variables, veut que l'on écrive les noms de fonctions en minuscule en séparant les mots par des underscores.

L'appel d'une fonction se fait en écrivant dans le programme le nom de la fonction avec les parenthèses :

```
...
ma_fonction ()
...
```

La fonction suivante affiche par exemple "Bonjour !" dans la console :

```
>>> def  f () :
...       print (" Bonjour ! ")
...
>>> f ()
Bonjour  !
```

Si la fonction retourne un résultat, il est spécifié à l'aide du mot-clé **return** Le corps d'une fonction peut contenir plusieurs **return**, mais un seul sera lu : dès que ce mot clé est rencontré, l'exécution de la fonction se termine et les instructions suivantes sont ignorées.

```
def  ma_fonction () :
     instruction_1
     ...
```

```
    instruction_n
    return  resultat
```

Pour récupérer le résultat, nous l'affectons à une variable de manière classique :

```
x = ma_fonction ()
```

Si la fonction retourne plusieurs résultats, on procédera ainsi :

```
def  ma_fonction () :
    instruction_1
    ...
    instruction_n
    return  resultat_1 ,  resultat_2 ,  ... , resultat_n

x_1 , x_2 , ...  x_n  = ma_fonction ()
```

Les paramètres qui peuvent être spécifiées en entrée de la fonction se mettent entre les parenthèses. Ces paramètres d'entrée n'ont pas de type défini. Selon les opérations effectuées, on peut appliquer la fonction à des réels, des matrices, des chaînes de caractères, ect.

Voici le cas le plus général de syntaxe pour définir une fonction

```
def  ma_fonction ( parametre_1 ,   ... , parametre_n ):
    instruction_1
    ...
    instruction_n
    return  resultat_1 ,  .. resultat_n
```

A noter que, lorsque la fonction renvoie plusieurs sorties, le type de la sortie est ce que l'on appelle un **tuple** Attention ce n'est pas une liste. En effet, la seule opération que l'on peut faire sur ce type d'objet appelé tuple et d'en extraire ses coordonnées (**tuple[i]**).

8.1.2 Appel de la fonction

Il est possible de définir des paramètres ayant une valeur par défaut. Si ces paramètres ne sont pas spécifiés lors de l'appel à la fonction, ce seront les valeurs par défaut qui seront utilisées. La seule contrainte est que les paramètres ayant une valeur par défaut doivent être positionnés à la fin de la liste des paramètres :

```
def  ma_fonction (a ,  b , c =2 ,d =3) :
```

Cette fonction pourra être appelée de trois façons différentes :

```
>>>  ma_fonction (0 ,1)        #a =0 , b =1 , c =2 , d =3
>>>  ma_fonction (0 ,1 ,4)     # a =0 , b =1 , c =4 , d =3
>>>  ma_fonction (0 ,1 ,4 ,5)  #a =0 , b =1 , c =4 , d =5
```

Lors de l'appel de la fonction il convient alors de spécifier la valeur des différents paramètres.Si la fonction attend plusieurs paramètres, lors de l'appel, les paramètres doivent être passés dans le même ordre que dans la définition de la fonction. Sinon pour s'assurer d'éviter toute erreur on peut aussi nommer les paramètres lors de l'appel et ainsi les mettre dans l'ordre que l'on veut.

Par exemple, la fonction suivante retourne la division d'un réel par un autre :

```
>>>  def  division ( nombre_1 ,  nombre_2 ):
...       resultat   = nombre_1 / nombre_2
...       return  resultat
...
>>>  x = division (4 ,2)  # effectue  la  division  de  4  par  2
>>>  x
2
>>>  x = division (2 ,4)  # effectue  la  division  de  2  par  4
>>>  x
0 ,5
>>>  x = division ( nombre_2 =2 , nombre_1 =4)  # effectue  la  division  de  4  par  2
>>>  x
2
```

A noter que Python ne permet pas que plusieurs fonctions portent le même nom dans un programme.

8.2 Variables globales et locales

Par défaut :
les variables définies à l'intérieur d'une fonction n'existent qu'à l'intérieur de la fonction ;
les variables définies à l'extérieur d'une fonction sont utilisables à l'intérieur de la fonction si elles ont été déclarée avant la fonction ;
les variables définies à l'extérieur d'une fonction ne sont pas modifiables dans cette fonction.

On désigne également de locale les variables dont la portée se limite à la fonction dans laquelle elles sont définies.

A l'inverse, une variable globale est visible dans l'ensemble du programme.

Observons avec quelques exemples qu'une variable définie avant une fonction est lisible dans la fonction mais pas modifiable et que les variable de la fonction ne sont visibles que dans la fonction :

```
>>> x = 0          # variable locale
>>> def test1 () :
...     y = x + 2
...     print (y)
...
>>> test1 ()
2
>>> y
NameError : name 'y ' is not defined     # y n'existe que dans la fonction
>>> def test2 () :
...     x = 3    # modification de la variable locale
...
>>> test2 ()
>>> x
0 # la variable globale n'est pas modifiée
```

Si l'on souhaite modifier une variable globale à l'intérieur d'un programme, il faut utiliser l'instruction **global**:

```
>>> x = 1
>>> def test () :
...     global x
...     x = 2
...     print (x)
...
>>> test ()
2
>>> x
2
```

Mais on évitera d'utiliser des variables globales dans les fonctions car cela revient à faire passer un paramètre caché à la fonction. On préférera écrire explicitement les entrées/sorties de la fonction.

Chapitre 9

Modules et librairies Python

9.1 Importer un module, une librairie

Un grand nombre de fonctions ou programmes classiques fréquemment utilisés ou particulièrement utiles dans certains domaines sont déjà prédéfinies et rangés dans ce que l'on appelle des modules externes. Par exemple, les fonctions racines carrées, exponentielle, ect sont déjà définies et rangés dans le module maths. On peut ainsi les utiliser sans avoir à se préoccuper de comment les définir. Ces modules peuvent ensuite être regroupés dans ce que l'on appelle des librairies (aussi appelées package ou paquet ou encore bibliothèques) qui regroupent donc des modules qui sont par exemple fréquemment utilisés dans un domaine particulier ou pour une application particulière. La multitude de librairies disponibles sur Python est ce qui en fait une de ses grandes forces.

On fera ainsi appel à ces modules ou librairies, suivant les fonctions que nous aurons besoin d'utiliser.
Afin de pouvoir accéder à ces fonctions il faudra au préalable importer ces modules et/ou librairies. Il est aussi possible de créer ses propres modules (modules internes) dans lesquels on peut ranger des fonctions ou programmes que l'on a créé. Nous nous restreindrons en ECG à l'utilisation de modules externes.

9.1.1 Importer un module

Pour charger un module externe, il faut utiliser le mot clé **import** suivi du nom du module.

```
import module
```

Il est alors ensuite possible d'utiliser l'ensemble des fonctions définies dans celui-ci.
Ainsi, suite à l'import du module **math** , on peut utiliser la fonction racine carré (**sqrt()**)

```
>>> import math
>>> math . sqrt (4)
2
```

Si l'on souhaite utiliser qu'une seule fonction d'un module externe sans tout importer, on peut tout simplement écrire :

```
from module import fonction
```

L'avantage c'est que l'on appelle alors directement la fonction pour l'utiliser (la syntaxe est plus simple que dans le cas précédent) :

```
>>> from math import sqrt
>>> sqrt (4)
2
```

Si l'on souhaite utiliser cette syntaxe avec plusieurs fonctions, le plus simple est d'utiliser une astérisque qui importe l'ensemble des fonctions du module

```
from module import *
```

En fait, l'importation d'un module peut se réaliser selon l'une des deux syntaxes suivantes : **from module import *** ou **import module as mod**
La première syntaxe charge le module, l'exécute et place toutes les fonctions du module dans l'espace des noms communs. Si deux modules possèdent des fonctions portant le même nom, seule la fonction du dernier module importé sera exécutable (puisque son importation aura écrasé la précédente). C'est pourquoi, on préférera toujours la seconde syntaxe, qui réalise une importation en notation pointée. Avec ce type de syntaxe, les fonctions du module ne sont pas importées dans l'espace des noms communs mais dans un espace dédié. Pour y accéder, il faudra en revanche préciser dans quel espace propre il faut aller chercher la fonction, selon la syntaxe suivante : **import math as mt**, où on importe ici le module math, en nommant l'espace dédié mt.

```
>>> import math as mt
>>> mt . sqrt (4)
2
```

9.1.2 Importer une librairie

La syntaxe d'import d'une librairie/package (qui contient plusieurs modules) ou d'un module d'un package est très similaire à celle d'un module ou d'une fonction d'un module :

```
import package
from package . sous - package import module
```

Voici un exemple d'import de package :

```
import geometrie
from geometrie import primitive
from geometrie . primitive import point
```

Par convention, on place tous les imports de modules au début d'un programme, même s'il n'interviennent que plus loin dans le programme.

9.2 Modules standards

Nous listons ici quelques uns des modules standards qui peuvent être couramment utilisés. Nous entendons par "module standard" un module qu'il n'est pas nécessaire d'installer (il est présent avec la version par défaut de Python).

Module	Utilisation
os	Manipulation de chemins et fichiers
glob	Lister des fichiers avec une syntaxe Unix
subprocess	Exécuter un programme externe
sys	Informations sur l'environnement
time	Heure courante, fuseaux horaires
datetime	Gestion des dates et durées
collections	Etend les données de listes, dictionnaires, tuples...
itertools	Manipulations sur les itérables
math	Fonctions mathématiques
decimal	Manipulation des chiffres à virgule
fractions	Manipulation des fractions
random	Nombres aléatoires

Table 9.1 Quelques modules courants

Le site https://pypi.python.org qui référence des milliers d'autres modules.

9.3 Exemple de la librairie math

Pour disposer des fonctions mathématiques usuelles, la librairie d'origine du python se nomme math. Ce module contient notamment les fonctions usuelles (exponentielle, logarithme, racine, etc) et les constantes usuelles (pi, e, etc).

Commande	Utilisation
`math.exp()`	exponentielle
`math.log()`	logarithme en base naturelle
`math.sqrt()`	racine carrée
`math.cos(),math.sin(),math.tan()`	cosinus, sinus, tangente, ...
`math.oor(3.5)`	partie entière, donne ici 3
`math.ceil(-7.6)`	entier immédiatement supérieur, ici -7
`math.fmod(4.7, 1.5)`	modulo, ici 0.2
`math.factorial(4)`	factorielle 4, donc 24 (uniquement pour les entiers positifs)
`math.fsum(l)`	fait la somme des éléments d'une liste l, à préférer à sum, moins d'erreurs d'arrondis (comparer math.fsum([0.01 for i in range(100)]) et sum([0.01 for i in range(100)]))
`math.isinf(x)`	teste si x est infini (inf) et renvoie True si c'est le cas
`math.isnan(x)`	teste si x est nan (Not a Number) et renvoie True si c'est le cas

Table 9.2 Quelques fonctions du module maths

Voici un exemple d'application :

```
>>> from math import *
>>> pi
3.141592653589793
>>> exp (1e -5)
1.00001000005
>>> log (10)
2.302585092994046
>>> log (1024 , 2)
10.0
>>> cos ( pi /4)
0.7071067811865476
```

9.4 Librairies utilisées en ECG

Librairie	Utilisation
Numpy	Manipulation de matrices ou tableaux multidimensionnels et fonctions maths opérant sur ces tableaux
Panda	Manipulation de données à analyser
Matplotlib	Tracer des graphiques et visualiser des données (histogramme, diagramme circulaire, ect)

Table 9.3 Librairies utiles en ECG

9.5 Installer une librairie particulière

La meilleure façon d'installer des librairies sous Python est de le faire en utilisant son système de gestion de paquets : pip (installé automatiquement dans les dernières versions par défaut). Pip permet d'installer simplement un paquet de la manière suivante

```
pip install paquet
```

L'avantage de ce système, c'est que pip se charge tout seul de télécharger le paquet ainsi que ses dépendances (en se connectant au site http://pypi.python.org/pypi) où chaque développeur Python est libre d'enregistrer ses paquets, en respectant quelques conditions, dont le renseignement de dépendances.

De plus, pip permet de choisir la version que l'on souhaite installer.

Par exemple, la version 1.5 du paquet :

```
pip install paquet ==1.5
```

Il permet de mettre à jour un paquet avec la commande upgrade :

```
pip install paquet -- upgrade
```

Mais aussi de désinstaller les paquets devenus inutiles avec la commande uninstall :

```
pip uninstall paquet
```

Chapitre 10

La librairie NumPy

Ce module permet la manipulation de tableaux (appelé array en anglais). Nous travaillerons essentiellement sur des tableaux à 1 dimension (aussi appelés vecteurs) ou à 2 dimensions (aussi appelées matrices).

10.1 Introduction

Le module numpy est la boîte à outils indispensable pour faire du calcul scientique avec Python. Elle a commencé à être développée dans les années 1990 par Travis Oliphan.

Pour modéliser les vecteurs, matrices et, plus généralement, les tableaux à n dimensions, numpy fournit le type ndarray.

On note des différences majeures avec les listes qui pourraient elles aussi nous servir à représenter des vecteurs :

Les tableaux numpy sont homogènes, c'est-à-dire constitués d'éléments du même type (entiers, flottants, chaînes de caractères, etc).

La taille des tableaux numpy est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme on le ferrait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ces contraintes sont en fait des avantages :

Le format d'un tableau numpy et la taille des objets qui le composent étant fixé, l'accès à ses éléments se fait en temps constant.

Les opérations sur les tableaux sont optimisées en fonction du type des éléments, et sont beaucoup plus rapide qu'elles ne le seraient sur des listes équivalentes.

Lorsque nous utiliserons cette bibliothèque nous l'importerons dans un espace de nom dédié (cf. Chapitre 9) que nous appellerons `np`.

```
import numpy as np
```

10.2 Constantes et fonctions mathématiques classiques

La bibliothèque numpy contient le module maths et de ce fait toutes les fonctions classiques (exponentielle, logarithme, etc). Elle contient aussi les valeurs de π et de exp 1 au travers des commandes **`np.pi`** et **`np.e`**.

```
>>> import numpy as np
>>> np . pi
3.141592653589793
>>> np .e
2.718281828459045
>>> np . cos (0.5)
0.87758256189037276
>>> np . exp (3)
20.085536923187668
```

Le principal intérêt de la librairie Numpy par rapport à celle Math est qu'elle permet la manipulation de vecteurs et de matrices.

10.3 Les vecteurs

10.3.1 Création de vecteurs et affichages

On utilisera la commande **np.array** en rentrant les coefficients à la main.
Voici un exemple :

```
>>> T= np . array ([1 ,2, 3]])
>>> T
array ([1 , 2, 3])
>>> type (T)
< class ' numpy . ndarray '>
>>> R= list (T)
>>> R
[1 ,2 ,3]
```

On peut ainsi transformer un array en liste. Attention ce n'est plus le même type de données et certaines opérations sur un type ne marche pas avec l'autre

10.3.2 Vecteurs particuliers

Vecteurs avec des zéros ou des uns

La commande **np.zeros(n)** crée un vecteur de taille n contenant que des 0.
La commande **np.ones(n)** crée un vecteur de taille n contenant que des 1.

np.arange

```
>>> T = np . arange (2 ,12 ,2)
>>> T
array ([ 2 ,4 ,6 ,8 ,10])
>>> type (m)
< class ' numpy . ndarray '>
```

```
>>> a= range (2 ,12 ,2)
>>> a
[2 , 4, 6, 8, 10]
>>> type (a)
< class ' range '>
```

Il faut noter la différence entre **np.arange()** et **range()** qui sont de types différents (array et liste) et pour lesquels les modalités d'utilisation et fonctionnalité sont différentes.
Contrairement à **range()**, **np.arange()** accepte des arguments qui ne sont pas des entiers

```
>>> T= np . arange (0 ,1.3 ,0.2)
>>> T
array ([ 0. , 0.2 , 0.4 , 0.6 , 0.8 , 1. , 1.2])
```

np.linspace

La commande **np.linspace()** permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments.

```
>>> np . linspace (2 ,12 ,5)
array ([ 2. , 4.5 , 7. , 9.5 , 12. ])
```

Si l'on ne précise pas le pas, il est de 1 par convention et, si on ne précise pas, le départ il vaut 0 par convention. Par contre il faut toujours préciser l'arrivée.

10.3.3 Manipulation des vecteurs

On va retrouver ici des commandes similaires à celles utilisées pour les listes.

Accès à un élément du tableau

Comme pour les listes, on accède à un élément du tableau par des crochets et les indices des éléments commencent à 0.

```
>>>   T=  np . array ([[1 ,2,  3])
>>>  T [0] #1er  coefficient
1
>>>  T [1]  # 2ème  coefficient
2
>>>  T [2]  # 3ème  coefficient
3
>>>  T [ -1]#  dernier   coefficient   = 3ème  coefficient
3
```

A noter que l'on peut aussi en utilisant des indices négatifs récupérer le dernier coefficient ou l'avant dernier coefficient.

On peut aussi extraire toute une séquence.
Soit T un vecteur.

Opérateur	**Signification**
`T[i]`	élément à la i-ème position
`T[i :j]`	sous-séquence de T, de la i-ème à la j-1-ème position
`T[i :j :k]`	sous-séquence de T, de la i-ème à la j-1-ème position avec un pas de k

Table 10.1 Opérations d'extraction sur les vecteurs

Attention, quand on parle de la ième position, comme les indices commence à 0, il s'agit en fait de la i+1 ème position dans notre façon de compter classique en démarrant à 1.

Les affectations se font de manière classique et peuvent se faire lorsque les objets de part et d'autre du = sont de même taille.

10.4 Les matrices

10.4.1 Création de matrices

La syntaxe est similaire à celle des vecteurs, en séparant les lignes par des virgules :

```
>>>  T=  np . array ([[1 ,2 , 3] , [4 , 5,  6]])
```

10.4.2 Matrices particulières

Commande	**Signification**
`np.zeros([n,p])`	matrices de n lignes et p colonnes avec que des coefficients égaux à zéro
`np.ones([n,p])`	matrices de n lignes et p colonnes avec que des coefficients égaux à un
`np.eye(n)`	matrice identité de taille n

Table 10.2 Quelques matrices particulières

10.4.3 Manipulation des matrices

Contrairement aux vecteurs, les coefficients d'une matrice sont déterminés par deux indices (numéro de ligne et de colonne dans cet ordre). Mais comme pour les vecteurs ou les listes, les indices commencent à 0 et peuvent prendre des valeurs négatives (-1 pour le dernier, -2 pour l'avant dernier, etc).

```
>>> T= np . array ([[1 ,2 , 3] , [4 , 5, 6]])
>>> T [0 ,1] # coefficient  première  ligne  et  deuxième  colonne
2
>>> T [1 ,2] # coefficient  deuxième  ligne  et  troisième  colonne
6
>>> T [ -1 , -1] # coefficient  dernière  ligne  et  dernière  colonne  =
    coefficient  deuxième  ligne  et  troisième  colonne
6
```

Soit T une matrice.

Opérateur	Signification
`T[i,j]`	élément à la i-ème ligne et j-ème colonne
`T[i :j, k :l]`	sous-séquence de T, de la i-ème à la j-1-ème ligne et de la k-ème à la l-1-ème colonne
`T[i :j, :]`	sous-séquence de T, de la i-ème à la j-1-ème ligne, toutes les colonnes incluses
`T[ : ,i :j]`	sous-séquence de T, de la i-ème à la j-1-ème colonnes, toutes les lignes incluses

Table 10.3 Opérations d'extractions sur les matrices

Attention, quand on parle de la ième position, comme les indices commence à 0, il s'agit en fait de la i+1 ème position dans notre façon de compter classique en démarrant à 1.

Les affectations se font de manière classique et peuvent se faire lorsque les objets de part et d'autre du = sont de même taille.

10.5 Opérations communes à tous les tableaux (vecteurs ou matrices)

10.5.1 Opérations classiques

Soit T et M des tableaux et a un réel.

Opérateur	Signification
`T + M`	somme coefficients par coefficients des tableaux T et M (qui doivent donc être de même taille)
`T * M`	produit coefficients par coefficients des tableaux T et M (qui doivent donc être de même taille)
`T/M`	division coefficients par coefficients du tableau T par M (qui doivent donc être de même taille)
`T+a`	ajout de a à tous les coefficients de T
`T*a`	multiplication de tous les coefficients de T par a
`T**a`	tous les coefficients de T sont élevés à la puissance a
`T/a`	division de tous les coefficients de T par a
`np.shape(M)`	taille du tableau
`np.dot(T,M)`	produit matriciel de T et M (à condition que les tailles des tableaux concordent)
`np.transpose(T)`	transposée de T

Table 10.4 Opérations classiques sur les tableaux

Voici un exemple de manipulation de vecteurs :

```
>>> import  numpy  as  np
>>> a = np . array ([1 ,2,  3 , 4])
>>> a  # a  est  un  tableau  d'entiers
```

```
array ([1 , 2, 3, 4])
>>> b = a * 1.5
>>> b # b est devenu un tableau de flottants
array ([ 1.5 , 3. , 4.5 , 6. ])
>>> c=a+ b
>>> c
array ([ 2.5 , 5. , 7.5 , 10. ])
```

10.5.2 Application de fonctions

Numpy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau. La syntaxe générale est **`np.nomfonction(*)`** où **`*`** peut être un entier, un réel, un vecteur, une matrice ou tout type d'objet auquel appliquer cette fonction a du sens.

En voici quelques unes.

Fonctions	**Signification**
`np.exp()`	fonction exponentielle
`np.log()`	fonction logarithme
`np.sqrt()`	fonction racine carrée
`np.oor()`	fonction partie entière
`np.abs()`	fonction valeur absolue
`np.shape()`	taille du tableau
`np.sum()`	somme des coefficients
`np.prod()`	produit des coefficients
`np.min()`	minimum des coefficients
`np.max()`	maximum des coefficients
`np.mean()`	moyenne des coefficients
`np.cumsum()`	somme cumulée des coefficients
`np.median()`	médiane des coefficients
`np.var()`	variance des coefficients
`np.std()`	écart type des coefficients

Table 10.5 Principales fonctions applicables sur des tableaux

Voici un exemple :

```
>>> import numpy as np
>>> T= np . arange (1 ,10 ,1)
>>> T
array ([1 , 2, 3, 4, 5, 6 , 7, 8, 9])
>>> np . sqrt (T )
array ([ 1. , 1.41421356 , 1.73205081 , 2. , 2.23606798 ,
 2.44948974 , 2.64575131 , 2.82842712 , 3. ])
```

A noter que lorsqu'il s'agit d'une matrice, on peut soit utiliser les fonctions suivantes sur la matrice entière (**`np.nomfonction(M)`**) ou demander à ce qu'elles soient appliquées pour chaque vecteur colonne indépendamment (**`np.nomfonction(M,2)`**) ou chaque vecteur ligne indépendamment (**`np.nomfonction(M,1)`**).

10.6 Le module Linalg

La librairie Numpy contient le module Linalg.

```
import numpy . linalg as al
```

Ce module contient les commandes suivantes qui pourront nous être utiles cette année :

Commande	Signification
`al.inv(A)`	calcul de l'inverse d'une matrice carrée A lorsqu'il existe
`al.matrix_power (A, n)`	calcul de la puissance n d'une matrice carrée A
`al.solve(A, B)`	résolution d'un système du type $Ax = B$.

Table 10.6 Deux commandes importantes

```
>>> T= np . array ([[1 ,2] , [4 , 5]])
>>> al . inv (T)# calcule l'inverse de T
array ([[ -1.66666667 ,  0.66666667] ,
        [ 1.33333333 ,  -0.33333333]])
>>> al . matrix_power (T ,3)  # calcule la puissance 3 de la matrice T
array ([[ 57 ,  78] ,
        [156 , 213]])
```

Résolvons le système : $\begin{aligned} x + 2y &= 1 \\ 3x + 5y &= 2 \end{aligned}$

```
>>> A= np . array ([[1 ,2] , [3 , 5]])
>>> B = np . array ([1 ,2])
>>> X = np . linalg . solve (A ,B )
>>> X
array ([ -1. ,  1.])
```

10.7 Le module random

La librairie Numpy contient le module random.

```
import  numpy . random  as  rd
```

Cette librairie permet de simuler la plupart des lois de probabilités usuelles et la commande générale prend la forme suivante : **`rd.nomdelaloi(*)`** où `*` représente les paramètres de la loi.

Commande	Signification
`rd.random()`	nombre tiré au hasard entre 0 et 1
`rd.randint(a,b)`	entier tiré au hasard entre a et b-1
`rd.random(N)`	N réalisation d'une loi $U([0, 1])$
`rd.binomial(n, p,N)`	N réalisations d'une loi $B(n, p)$
`rd.randint(a, b,N)`	N réalisations d'une loi $U([\![a, b-1]\!])$
`rd.geometric(p,N)`	N réalisations d'une loi $G(p)$
`rd.poisson(λ, N)`	N réalisations d'une loi $P(\lambda)$
`rd.normal(m, str, N)`	N réalisations d'une loi normale $N(m, str^2)$
`rd.exponential(λ, N)`	N réalisations d'une loi exponentielle $\varepsilon(\lambda)$
`rd.gamma(k, r, N)`	N réalisations d'une loi gamma $\Gamma(k, r)$

Table 10.7 Lois de probabilités usuelles

La sortie est de type **`array`** (sauf pour **`rd.random()`** et **`rd.randint(a,b)`** qui sont de type **`float`**). Si le nombre de réalisations N à effectuer n'est pas renseigné alors par défaut une seule réalisation sera faite. Au contraire si l'on souhaite créer un tableau ayant L lignes et C colonnes dont les coefficients seraient des réalisations d'une loi donnée alors il faut remplacer N par [L, C].

Voici un exemple :

```
>>> import  numpy . random  as  rd
>>> a= rd . binomial (10 ,0.5) #1 réalisation d'une binomiale B (10 , 0.5)
>>> a
4
>>> b= rd . binomial (10 ,0.5 ,5) #5 réalisation d'une binomiale B (10 , 0.5)
>>> b
```

```
array ([4 , 4, 6, 3, 6])
>>> c= rd . binomial (10 ,0.5 ,[2 ,3]) # tableau de taille 2*3 contenant des ré
    alisations d'une binomiale B (10 ,0.5)
>>> c
array ([[6 , 4, 8] ,
        [7 , 7, 7]])
>>> c= rd . binomial (10 ,0.5 ,[2 ,3])
>>> c
array ([[4 , 2, 1] ,
        [4 , 5, 4]])
>>> type (a)
< class ' int '>
>>> type (b)
< class ' numpy . ndarray '>
>>> type (c)
< class ' numpy . ndarray '>
```

Chapitre 11

La librairie Matplotlib

11.1 Introduction

Cette librairie Matplotlib permet toutes sortes de représentations de graphes 2D (et quelques unes en 3D).
Comme les graphiques font intervenir des vecteurs et des fonctions usuelles, ce module a besoin pour exister de celui numpy. Et c'est le module pyplot qui permet de tracer ces fonctions.

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

11.2 Dans la pratique

Pour tracer des fonctions, Python procède par interpolation de points (tracé de segments entre deux points donnés par l'utilisateur et déterminés par leur abscisse et leur ordonnée).

Lorsque l'on souhaite tracer une fonction, on utilisera ainsi la fonction `plt.plot` qui possède deux arguments d'entrée :

- un tableau 1D (vecteur x) contenant les valeurs des abscisses que l'on construit généralement à l'aide des commandes `np.arange` ou `np.linspace` (suivant que l'on veuille un pas fixe suffisamment petit ou un nombre de points donné susamment grand pour définir les abscisses des points à tracer de manière à obtenir une courbe lisse) ;
- un tableau 1D (vecteur y), de même longueur, contenant les valeurs des ordonnées (ainsi si l'on souhaite tracer la fonction f on pourra ainsi poser $y = f(x)$ où f est une fonction déjà dénie ou une que l'on a défini au préalable).

A noter que x et y peuvent être de type range ou liste, mais on privilégiera de préférence des données de type array.

On utilisera ensuite la commande `plt.plot` pour faire le tracé.
A noter que la fonction `plt.plot` seule ne permet pas l'affichage de la courbe à l'écran. Il faut lui ajouter à la suite la fonction `plt.show` qui remplit ce rôle.

Voici un exemple de tracé de fonction :

```
import numpy as np # on importe la librairie numpy
import matplotlib . pyplot as plt # on importe le sous - module pyplot
x = np . linspace (0 ,2* np . pi ,100)#on définit ici le vecteur des abscisses
    x
y = np . cos (x )#on définit ici le vecteur des ordonnées y
plt . plot (x ,y )#on réalise le tracé de la courbe y en fonction de x
plt . show () # on demande à Python d'afficher la courbe à l'écran
```

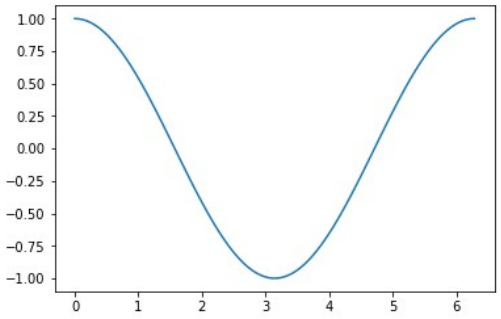


Figure 11.1 Tracé simple d'un cosinus.

11.3 Mise en forme des graphiques

Les commandes ci dessous ne sont pas exigibles à votre programme.
Cependant comme elles ne sont pas compliquées à comprendre et à utiliser, je vous les présente ici et nous aurons l'occasion de les utiliser pendant l'année.

11.3.1 Autour du repère

Autres paramètres que l'on peut définir :

Commande	**Signification**
`plt.xlim(xmin, xmax)`	restreint la représentation graphique à l'intervalle [xmin, xmax] en abscisse
`plt.ylim(ymin, ymax)`	restreint la représentation graphique à l'intervalle [ymin, ymax] en ordonnée
`plt.xlabel('nom axe abscisses')`	donne un titre à l'axe des abscisses
`plt.ylabel('nom axe ordonnées')`	donne un titre à l'axe des ordonnées
`plt.title('nom du graphique')`	donne un titre au graphique
`plt.legend()`	affiche la légende dénie par label='nom' dans plt.plot
`plt.axis('equal')`	pour que le repère soit orthonormé
`plt.grid()`	ajoute une grille au graphique
`plt.save('nom du fichier')`	sauvegarde la figure dans le dossier courant

Table 11.1 Commandes pour améliorer le graphique

Ce second exemple améliore le tracé précédent.

```
import numpy as np
import matplotlib . pyplot as plt
x = np . linspace (0 ,2* np . pi ,100)
y = np . cos (x )
plt . plot (x ,y ,label = ' cos (x) ')
plt . ylim ( -2 ,2)
plt . title ( ' tracéde la fonction cosinus ')# on ajoute un titre au
    graphique
plt . xlabel ( ' Angle( en radians ) ')# on ajoute un titre à l'axe x
plt . legend () # affiche la lé gende définie par label
plt . show ()
plt . save ( ' cosinus ')# sauvegarde de la figure
```

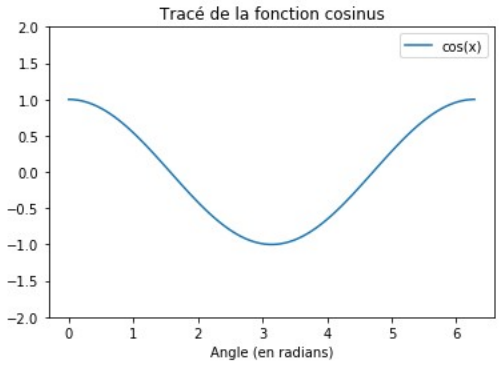


Figure 11.2 Tracé amélioré d'un cosinus.

11.3.2 Couleurs et style

On peut aussi utiliser les commandes suivantes pour changer la couleur de la courbe :

Couleur	**Syntaxe**
Rouge	`'r'`
Vert	`'g'`
Bleu	`'b'`
Cyan	`'c'`
Magenta	`'m'`
Jaune	`'y'`
Noir	`'k'`

Ou le style de la courbe :

Style	**Syntaxe**
Pointillé	`''`
Lignes en pointillés	`' :'`
Traits/points	`'-.'`
Cercle	`'o'`
Triangle	`'v'`
Carré	`'s'`
Signe +	`'+'`
Croix	`'x'`

Pour cela on utilise la commande **`plt.plot(x, y,styleDuGraphe)`** où **`styleDuGraphe`** est une chaîne de caractères qui regroupe la couleur de la courbe, le marqueur de point et le style de liaison entre les points.

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
x  =  np . linspace (0 ,2* np . pi ,100)
y  =  np . cos (x )
plt . plot (x ,y ,r -. ', label = ' cos (x) ')# tracé  en  rouge  avec  des  pointillés
plt . ylim ( -2 ,2)
plt . title ( 'tracé de  la  fonction  cosinus ')#  on  ajoute  un  titre  au
    graphique
plt . xlabel ( ' Angle( en radians ) ')# on  ajoute  un  titre  à l'axe  x
plt . legend () # affiche  la légende  définie  par  label
plt . show ()
plt . save ( ' cosinus ')# sauvegarde  de  la  figure
```

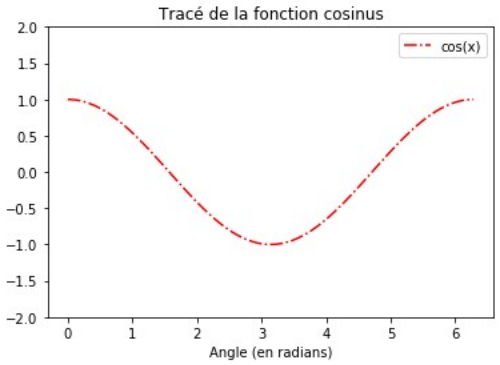


Figure 11.3 Tracé décoré d'un cosinus.

11.4 Tracé multiple

A nouveau, les commandes vues ici ne sont pas exigibles en ECG, mais bien utiles cependant en pratique.

11.4.1 Sur une même figure

Pour tracer plusieurs courbes sur le même graphique, on utilise la même syntaxe que pour une représentation.

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
x  =  np . linspace (0 ,2* np . pi ,100)# on  définit  ici  le  vecteur  des  abscisses
      x
y1 = np . cos (x )# on  définit  ici  le  1er  vecteur  des  ordonnées
y2 = np . sin (x )#on  dé finit  ici  le  2nd  vecteur  des  ordonnées
plt . plot (x ,y1 , 'b ', label = ' cos (x)\n')# Réalise le  tracé  de  cosinus  en
     bleu
plt . plot (x ,y2 , 'g ', label = ' sin (x)\n')# Réalise le  tracé  de  sinus  en  vert
plt . legend () #on  demande  à  Python  d'afficher  les  légendes  sur  la
plt . show () #on  demande  à  Python  d'afficher  les  courbes  à  l 'écran
```

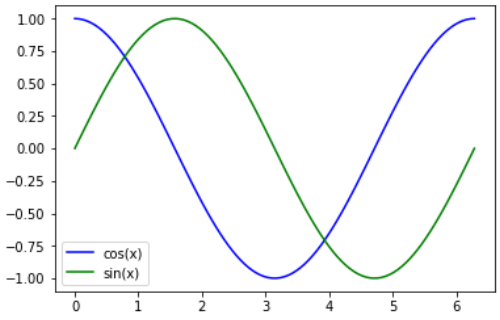


Figure 11.4 Tracés multiples sur une même figure.

11.4.2 Sur des figures différentes

On utilise la commande `subplot` qui permet de diviser une fenêtre graphique en sous-fenêtres. Trois paramètres sont à donner :

1. le nombre de ligne du découpage (1er paramètre)
2. le nombre de colonne du découpage (2ème paramètre)

3. le numéro de la case où on positionne la figure (on commence à numéroter les cases de la première ligne, puis celles de la seconde, ect).

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
x = np . linspace (0 ,2* np . pi ,100) on définit  ici  le  vecteur  des  abscisses
     x
y1 = np . cos ( x)# on dé finit  ici  le  1er  vecteur  des  ordonnées
t= np . linspace ( np .pi ,3* np . pi ,100)on dé finit  ici  le  vecteur  des
    abscisses  t
y2 = np . sin (t)#on dé finit  ici  le  2nd  vecteur  des  ordonnées
plt . subplot (211) # on découpe  la  feuille  en  2  lignes  et  1  colonne  et  on
    place  le  graphique  à  la  1è re ligne  et  1è re colonne
plt . plot (x ,y1 , 'bo label = ' cos ')# on ré alise  le  tracé  de  la  1è re
    fonction
plt . legend ()
plt . subplot (212) # on dé coupe  la  feuille  en  2  lignes  et  1  colonne  et  on
    place  le  graphique  à  la  2è me ligne  et  1è re colonne
plt . plot (t ,y2 , 'r -- label = ' sin ')#on réalise  le  tracé  de  la  seconde
    fonction
plt . legend ()
plt . show () #Python  d'afficher  les  courbes  à  l 'écran
```

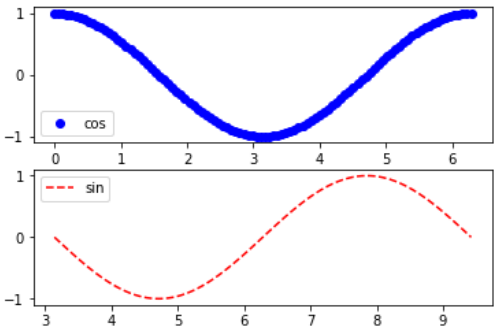


Figure 11.5 Tracés multiples sur des figures différentes.

11.5 Représentation statistiques

Pour faire des représentations de données statistiques sous forme d'histogramme, de diagrammes en bâton ou de boites à moustaches, on utilisera les commandes suivantes :

Commande	Signification
`plt.hist`	pour tracer un histogramme
`plt.bar`	pour tracer un diagramme en bâton
`plt.boxplot`	pour tracer des boites à moustaches

Table 11.2 Représentations de données statistiques

11.5.1 Histogrammes

Création d'un histogramme

La commande **`plt.hist(valeurs, bins= *, range=**)`** permet de tracer un histogramme, elle prend trois arguments :

- le vecteurs **`valeurs`** qui contient la série de données dont on veut tracer l'histogramme ;

- le **paramètre `range`** permettant de définir les valeurs min et max de l'intervalle global sur lequel tracer l'histogramme, par défaut elles valent le minimum et le maximum de la série de valeurs ;
- le **paramètre `bins`** permettant de définir les classes de l'histogramme :
  — soit en donnant le nombre n de classes en rentrant un entier. Dans ce cas Python se charge alors de découper l'intervalle allant de la plus petite valeur à la plus grande (données par **`range`** ou celles par défaut) en n classes de longueur égales.
  Par exemple, avec **`bins=6`** on aura 6 intervalles.
  — soit en donnant le nombre de classes et leurs largueurs, en rentrant une liste, un tableau ou une séquence dénissant les bornes des classes (valeur de gauche incluse et celle de droite exclue (sauf la dernière)).
  Les classes peuvent être de longueurs inégales (dans ce cas on rentrera les bornes à la main) ou de longueurs égales (dans ce cas on pourra utiliser les commandes **`range`** ou **`np.linspace`** ou **`np.arange`** pour les définir).
  Par exemple, avec **`bins = [0,1,4,6]`** on aura trois classes d'intervalles de longueur inégales [0,1[; [1,4[et [4,6[.
  Alors qu'avec **`np.arange(-0.5, 5.5)`** on aura 5 classes de longueurs égales [-0.5, 0.5[, [0.5, 1.5[, [1.5, 2.5], [2.5, 3.5[, [3.5, 4.5 [.

Voici l'histogramme le plus basique que l'on puisse tracer.

```
import  numpy  as  np
import  numpy . random  as  rd
import  matplotlib . pyplot  as  plt
# Tableau  de  10  valeurs  comprises  entre  0 et  10
valeurs  = np . random . randint (0 ,11 , 100)
# Création  d 'un tableau  avec  les  intervalles  centrés  sur  la  valeur  entiè
    re
inter  = np . arange ( -0.5 ,11.5)
# Création  de  l'histogramme
plt . hist ( valeurs ,bins = inter )
plt . show ()
```

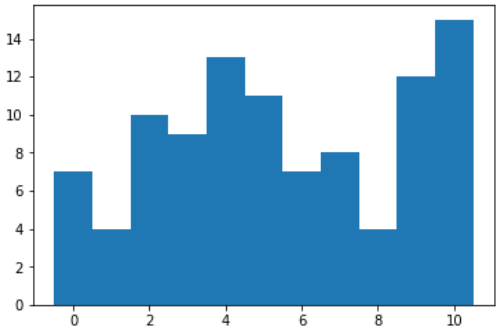


Figure 11.6 Histogramme.

Autour des axes

Paramètre	Signification
`plt.xlabel()`	titre des abscisses
`plt.ylabel()`	titre des ordonnées
`plt.title()`	titre de l'histogramme
`plt.xticks()`	affiche le centre des intervalles

Table 11.3 Paramétrisation des axes

```
import numpy as np
import numpy . random as rd
import matplotlib . pyplot as plt
# Tableau de 10 valeurs comprises entre 0 et 10
valeurs = np . random . randint (0 ,11 , 100)
# Création d 'un tableau avec les intervalles centrés sur la valeur enti è
    re
inter = np . arange ( -0.5 ,11.5)
# Création de l'histogramme
plt . hist ( valeurs ,bins = inter )
plt . xlabel ( ' Valeurs ')
plt . xticks ( np . arange (01,1) )
plt . ylabel ( ' Nombres ')
plt . title ( ' Histogramme ')
plt . show ()
```

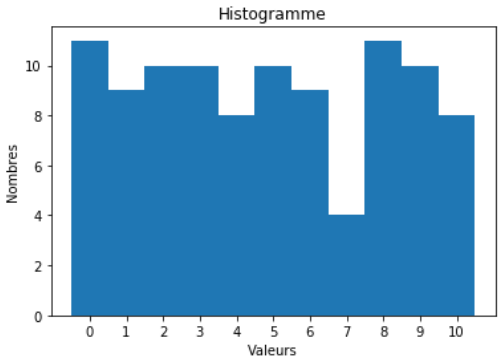


Figure 11.7 Histogramme avec titres.

On voit ici que la lecture serait plus claire si les bâtons étaient espacés. Nous allons voir dans le paragraphe suivant comment améliorer le visuel de l'histogramme et le personnaliser.

Personnalisation de l'histogramme

Paramètre	Signification
`density=True`	Trace les fréquences plutôt que les nombres en ordonnée (somme vaut 1).
`histtype='bar'`	défini le type d'histogramme à afficher
`align='mid'`	Alignement de la barre par rapport au centre de l'intervalle : 'left', 'mid' ou 'right'.
`orientation='vertical'`	Orientation des barres : 'horizontal' ou 'vertical'.
`rwidth=0.8`	Largeur relative de la barre par rapport à l'intervalle (ici 80 %).
`color='red'`	Couleur des barres.
`edgecolor='black'`	Couleur du cadre des barres.
`label='Série 1'`	Nom donné à la série. C'est le nom qui apparaît dans la légende.
`stacked=True`	Si il y a plusieurs séries de données, elles s'affichent empilées et non côte-à-côte.

Table 11.4 Personnalisation de l'histogramme

Grâce à la commande `rwidth` nous allons améliorer le visuel de notre histogramme précédent.

```
import numpy as np
import numpy . random as rd
import matplotlib . pyplot as plt
# Tableau de 10 valeurs comprises entre 0 et 10
valeurs = np . random . randint (0 ,11 , 100)
```

```
# Création d'un tableau avec les intervalles centrés sur la valeur enti è
    re
inter = np . arange ( -0.5 ,11.5)
# Création de l'histogramme
plt . hist ( valeurs ,bins = inter , rwidth =0.8)
plt . xlabel ( ' Valeurs ')
plt . xticks ( np . arange (01‡) )
plt . ylabel ( ' Nombres ')
plt . title ( ' Histogramme ')
plt . show ()
```

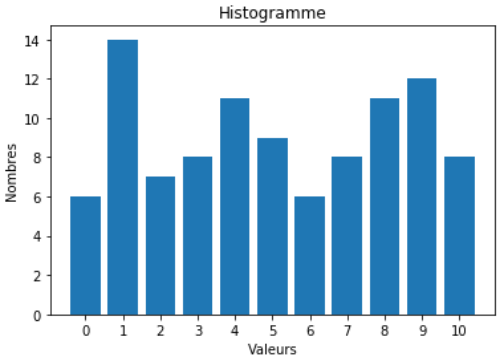


Figure 11.8 Histogramme amélioré.

Et voici un exemple d'histogramme avec deux séries de valeurs.

```
import numpy as np
import numpy . random as rd
import matplotlib . pyplot as plt
# Tableau de 100 valeurs comprises entre 0 et 10
valeurs = rd . randint (0 ,11 , 100)
# Tableau de 100 valeurs comprises entre 0 et 10
valeurs2 = rd . randint (0 ,11 , 100)
# Création d 'un tableau avec les intervalles centrés sur la valeur entiè
    re
inter = np . linspace ( -0.5 ,10.5 , 12)

plt . hist ([ valeurs ,valeurs2 ], color =[ ' red ',' blue '],bins = inter , rwidth
    =0.8 , stacked = True ,
          label =[ ' Valeurs 1 ', ' Valeurs 2 ']) # Création de l'histogramme
plt . xlabel ( ' Valeurs ')
plt . xticks ( np . arange (01‡) )
plt . ylabel ( ' Nombres ')
plt . title (" Histogramme " )
plt . legend ()
plt . show ()
```

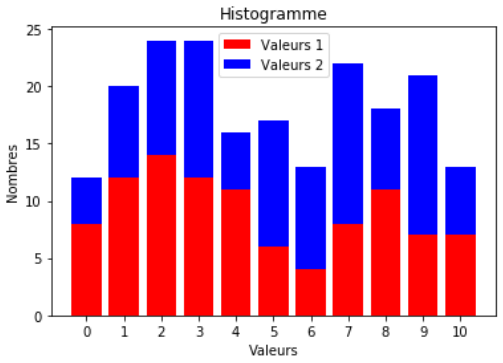


Figure 11.9 Histogramme évolué.

11.5.2 Diagramme en bâtons

Création d'un diagramme en bâton

La commande **`plt.bar(modalites, hauteurs, width)`** permet de tracer un diagramme en bâton elle prend trois arguments :

- le **vecteurs** modalites qui contient les modalités de la série dont on veut tracer le diagramme en bâton ;
- le **vecteurs** hauteurs contenant les effectifs (resp. les fréquences) des différentes modalités de la série dont on veut tracer le diagramme en bâton des effectifs (resp. fréquences) ;
- le **réel** `width` dénissant la largeur des bâtons (que l'on prend généralement égal à 0.05).

Prenons l'exemple de la série statistique suivante :

Valeur	1	2	3	4	5
Effectif	5	4	5	1	2

Voici le code Python pour tracer le diagramme en bâton associé à cette série :

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
modalite =[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10]
effectif  = [8 ,12 ,8 ,5 ,4 ,3 ,2 ,1 ,0 ,0]
width =0.05
plt . bar ( modalite ,effectif ,  width )
plt . show ()
```

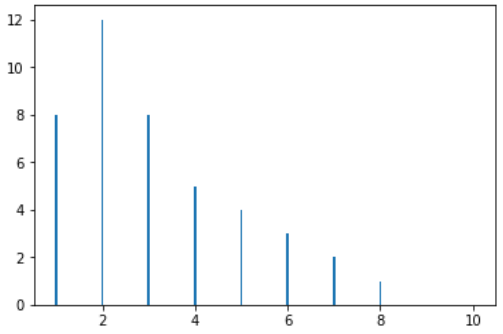


Figure 11.10 Diagramme en bâton.

Personnalisation du diagramme

Voici les paramètres que l'on peut ajouter

Paramètre	Signification
`color='b'`	pour choisir la couleur des bâtons, à mettre dans `plt.bar()`.
`plt.xlabel()`	titre des abscisses
`plt.ylabel()`	titre des ordonnées
`plt.title()`	titre du diagramme

Table 11.5 Personnalisation du diagramme

Voici un diagramme en bâton améliorée :

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
valeur =[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10]
effectif  = [8 ,12 ,8 ,5 ,4 ,3 ,2 ,1 ,0 ,0]
width =0.05
plt . bar ( valeur ,effectif ,  width ,  color = 'm ')
plt . title ( ' Diagramme en  bâ ton ')
plt . xlabel ( ' Modalit és ')
plt . ylabel ( ' Effectifs ')
plt . show ()
```

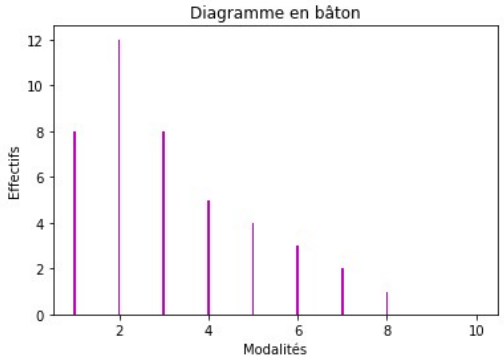


Figure 11.11 Diagramme en bâton améliorée.

11.5.3 Boîte à moustache

Création d'une boîte à moustache

La commande **`plt.boxplot(valeurs)`** permet de tracer un diagramme en bâton. Elle prend en entrée le vecteur **`valeurs`** qui contient l'ensemble des valeurs de la série statistique.

Prenons l'exemple de l'échantillon -1, 4,5,7,5,4,6,6,7,4,8,4,3,9,8,7,13.

Voici le code Python pour tracer la boîte à moustache associée à cet échantillon (comme pour les diagrammes en bâton on peut ajouter un titre) :

```
import  numpy  as  np
import  matplotlib . pyplot  as  plt
valeurs =[ -1 , 4 ,5 ,7 ,5 ,4 ,6 ,6 ,7 ,4 ,8 ,4 ,3 ,9 ,8 ,7 ,13]
plt . boxplot ( valeurs )
plt . title ( 'Boîte à  moustache ')
plt . show ()
```

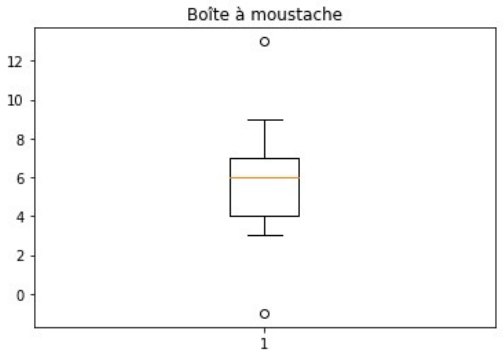


Figure 11.12 Boîte à moustache.

Représenter plusieurs boîte à moustache

Voici un exemple avec trois boîtes à moustache représentée sur le même graphique. Afin de les distinguer, on peut ajouter la commande **`plt.xticks()`** qui permet d'ajouter une étiquette sous chaque boîte à moustache.

```
serie1  = [1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9]
serie2  = [15 ,16 ,17 ,18 ,19 ,20 ,21 ,22 ,23 ,24 ,25]
serie3  = [5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13]

noms  = [ 'Série  1 ','Série  2 ','Série  3 ']

data  =[ serie1 , serie2 ,  serie3 ]

plt . boxplot ( data )

plt . xticks ([1 ,2 ,3] ,noms )

plt . title ( ' Comparaison de  3  boîtes  à  moustache ')

plt . show ()
```

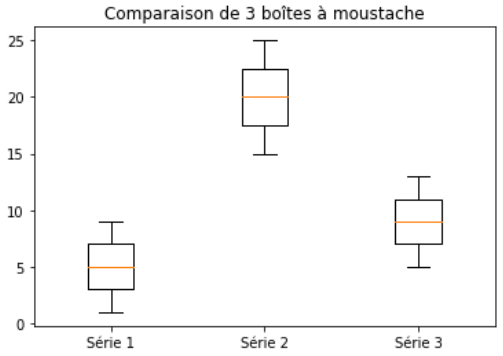


Figure 11.13 Plusieurs boîtes à moustache.

Chapitre 12

Dans la pratique

Comment faire en pratique pour écrire un programme ?

L'analyse qui permet de créer un algorithme et la programmation ensuite, sont deux phases différentes qui nécessitent de la pratique avant de devenir évidentes.

12.1 Réfléchir à un algorithme sur papier

12.1.1 On analyse les données

Je vous conseille fortement de démarrer sur papier, en identifiant tout d'abord quelles sont les données que l'on a à traiter en entrée et quelles sont les données que l'on s'attend à trouver en sortie et ce qui lie chaque données aux autres.

12.1.2 Résoudre à la main

On commence par une résolution du problème, en réalisant les transformations et calculs sur notre échantillon de problème, en fonctionnant par étapes.

À chaque étape, on note :

- quelles sont les étapes pertinentes, sur quels critères elles ont été choisies ;
- quelles sont les séquences d'opérations que l'on a répété.

Lorsque l'on tombe sur des étapes complexes, on découpe en sous-étapes, éventuellement en les traitant séparément comme un algorithme de résolution d'un sous-problème. Le but est d'arriver à un niveau de détails suffisamment simple ; soit qu'il s'agisse d'opérations très basiques (opération sur un texte, expression de calcul numérique. . .), soit que l'on pense/sache qu'il existe déjà un outil pour traiter ce sous-problème (calcul de sinus pour un angle, opération de tri sur une séquence de données. . .).

Normalement, au cours de ces opérations, on a commencé à nommer les données et les étapes au fur et à mesure qu'on en a eu besoins.

12.1.3 Formaliser

Une fois qu'on a un brouillon des étapes, il faut commencer à mettre en forme et à identifier les constructions algorithmiques connues et les données manipulées :

- Boucles (sur quelles informations, condition d'arrêt).
- Tests (quelle condition).

Informations en entrée, quel est leur type, quelles sont les contraintes pour qu'elles soient

- valides et utilisables, d'où viennent-elles :
- déjà présentes en mémoire,
- demandées à l'utilisateur,
- lues dans des fichiers ou récupérées ailleurs (sur l'Internet par exemple).

- Calculs et expressions : quel genre de données sont nécessaires, y-a-t-il des éléments constants à connaître, des résultats intermédiaires à réutiliser.
- Stockage des résultats intermédiaires.
- Résultat final, qu'en fait-on :
  - retourné dans le cadre d'une fonction,
  - affiché à l'utilisateur,
  - sauvegardé dans un fichier.

12.2 Passer de l'idée au programme

Le passage de l'idée puis de l'algorithme au code dans un programme, est relativement facile en Python car celui-ci est très proche d'un langage d'algorithmique.

Les **noms** des choses que l'on a manipulés vont nous donner des `variables`.
Les **tests** vont se transformer en **`if condition`**:
Les **boucles sur des séquences d'informations** vont se transformer en **`for variable in sequence:`**
Les **boucles avec expression de condition** vont se transformer en while conditions:
Les **séquences d'instructions qui se répètent** en différents endroits vont se transformer en **`fonctions`**.
Le **retour de résultat** d'une séquence (fonction) va se traduire en **`return variable`**.