

# Python pour les économistes

*Ewen Gallic*

*Octobre 2018*



# Table des matières

<b>Liste des tableaux</b>	<b>5</b>
<b>Table des figures</b>	<b>7</b>
<b>Propos liminaires</b>	<b>9</b>
0.1 Objectifs . . . . .	9
0.2 À qui s'adressent ces notes ? . . . . .	9
<b>1 Introduction</b>	<b>11</b>
1.1 Historique . . . . .	11
1.2 Versions . . . . .	11
1.3 Espace de travail . . . . .	13
1.4 Les variables . . . . .	21
1.5 Les commentaires . . . . .	24
1.6 Les modules et les packages . . . . .	24
1.7 L'aide . . . . .	26
<b>2 Types de données</b>	<b>29</b>
2.1 Chaînes de caractères . . . . .	29
2.2 Valeurs numériques . . . . .	40
2.3 Booléens . . . . .	44
2.4 Objet vide . . . . .	44
2.5 Dates et temps . . . . .	45
<b>3 Structures</b>	<b>57</b>
3.1 Listes . . . . .	57
3.2 N-uplets (Tuples) . . . . .	63
3.3 Ensembles . . . . .	64
3.4 Dictionnaires . . . . .	68
<b>4 Opérateurs</b>	<b>75</b>
4.1 Opérateurs arithmétiques . . . . .	75
4.2 Opérateurs de comparaison . . . . .	78
4.3 Opérateurs logiques . . . . .	82
4.4 Quelques fonctions . . . . .	85

4.5	Quelques constantes . . . . .	86
4.6	Exercice . . . . .	86
<b>5</b>	<b>Chargement et sauvegarde de données</b>	<b>89</b>
5.1	Charger des données . . . . .	90
5.2	Exporter des données . . . . .	96
<b>6</b>	<b>Conditions</b>	<b>101</b>
6.1	Les instructions conditionnelles <code>if</code> . . . . .	101
6.2	Les instructions conditionnelles <code>if-else</code> . . . . .	103
6.3	Les instructions conditionnelles <code>if-elif</code> . . . . .	103
6.4	Exercice . . . . .	104
<b>7</b>	<b>Boucles</b>	<b>107</b>
7.1	Boucles avec <code>while()</code> . . . . .	107
7.2	Boucles avec <code>for()</code> . . . . .	108
7.3	Exercice . . . . .	111
<b>8</b>	<b>Fonctions</b>	<b>113</b>
8.1	Définition . . . . .	113
8.2	Portée . . . . .	117
8.3	Fonctions <code>lambda</code> . . . . .	120
8.4	Retour de plusieurs valeurs . . . . .	121
8.5	Exercice . . . . .	121
<b>9</b>	<b>Introduction à Numpy</b>	<b>123</b>
<b>10</b>	<b>Manipulation de données avec Pandas</b>	<b>125</b>
10.1	Importation et exportation de données . . . . .	125
10.2	Sélection . . . . .	125
10.3	Filtrage . . . . .	125
10.4	Retrait des valeurs dupliquées . . . . .	125
10.5	Modification des colonnes . . . . .	125
10.6	Tri . . . . .	125
10.7	Jointures . . . . .	125
10.8	Agrégation . . . . .	125
10.9	Stacking et unstacking . . . . .	125
<b>11</b>	<b>Visualisation de données</b>	<b>127</b>
<b>12</b>	<b>Programmation parallèle</b>	<b>129</b>
<b>13</b>	<b>References</b>	<b>131</b>

# Liste des tableaux

2.3	Codes de formatages . . . . .	49
4.1	Opérateurs de comparaison . . . . .	78
4.2	Quelques fonctions numériques . . . . .	85
4.3	Quelques constantes intégrées dans Python . . . . .	86
5.1	Valeurs principales pour la manière d’ouvrir les fichiers. . . . .	90
5.2	Paramètres de la fonction <code>reader()</code> . . . . .	93



# Table des figures

1.1	Langages de programmation, de scripting et de balisage. . . . .	12
1.2	Python dans un terminal. . . . .	13
1.3	Fenêtre d'accueil d'Anaconda. . . . .	15
1.4	Console IPython. . . . .	15
1.5	Spyder. . . . .	17
1.6	Jupyter. . . . .	18
1.7	Un notebook vide. . . . .	18
1.8	Cellule évaluée. . . . .	19
1.9	Cellule textuelle non évaluée. . . . .	21





# Propos liminaires

Ces notes de cours ont été réalisées dans le cadre d'un enseignement d'introduction à Python adressé à des étudiants du parcours Économétrie et Big Data de [l'École d'Économie d'Aix-Marseille / Aix-Marseille School of Economics \(AMSE\)](#) d'Aix-Marseille Université.

## 0.1 Objectifs

Cet ouvrage a pour but l'initiation au langage de programmation Python, afin d'être capable de s'en servir de manière efficace et autonome. Le lecteur peut exécuter tous les exemples fournis (et est vivement encouragé à le faire). Des exercices viennent clore certains chapitres, pour mieux s'appropriier les notions couvertes au fur et à mesure de la lecture.

Bien évidemment, Python étant un langage très vaste, ces notes ne sauraient et n'ont pas pour vocation à être exhaustives de l'utilisation de ce langage informatique.

## 0.2 À qui s'adressent ces notes ?

Dans un premier temps, cet ouvrage s'adresse aux débutants qui souhaitent apprendre les bases en Python. Il est à destination des étudiants de l'AMSE mais pourrait intéresser des individus ayant une approche de la donnée à travers la discipline économique désirant découvrir Python.



# Chapitre 1

## Introduction

Ce document est construit principalement à l’aide de différentes références, parmi lesquelles :

- des livres : Briggs ([2013](#)), Grus ([2015](#)), VanderPlas ([2016](#)), McKinney ([2017](#)) ;
- des (excellents) notebooks : Navaro ([2018](#)).

### 1.1 Historique

Python est un langage de programmation multi plates-formes, écrit en C, placé sous une licence libre. Il s’agit d’un langage interprété, c’est-à-dire qu’il nécessite un interprète pour exécuter les commandes, et n’a pas de phase de compilation. Sa première version publique date de 1991. L’auteur principal, [Guido van Rossum](#) avait commencé à travailler sur ce langage de programmation durant la fin des années 1980. Le nom accordé au langage Python provient de l’intérêt de son créateur principal pour une série télévisée britannique diffusée sur la BBC intitulée “*Monty Python’s Flying Circus*”.

La popularité de Python a connu une croissance forte ces dernières années, comme le confirment les résultats de sondages proposés par [Stack Overflow](#) depuis 2011. Stack Overflow propose à ses utilisateurs de répondre à une enquête dans laquelle de nombreuses questions leur sont proposées, afin de décrire leur expérience en tant que développeur. [Les résultats de l’enquête de 2018](#) montrent une nouvelle avancée de l’utilisation de Python par les développeurs. En effet, comme le montre la Figure [1.1](#), 38.8% des répondants indiquent développer en Python, soit 6.8 points de pourcentage de plus qu’un an auparavant, ce qui fait de ce langage de programmation celui dont la croissance a été la plus importante entre 2017 et 2018.

### 1.2 Versions

Ces notes de cours visent à fournir une introduction à Python, dans sa version 3.x. En ce sens, les exemples fournis correspondront à cette version, non pas aux précédentes.

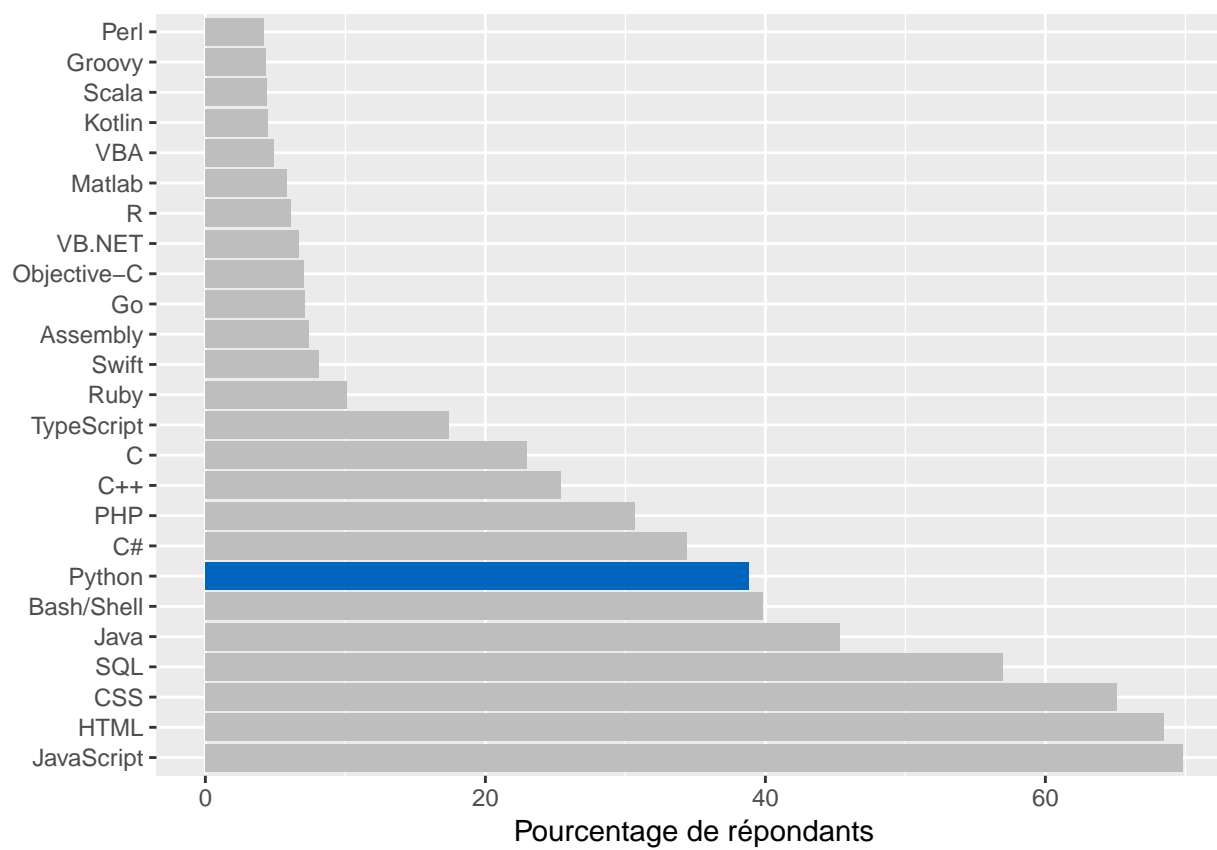


FIGURE 1.1 – Langages de programmation, de scripting et de balisage.

Comparativement à la version 2.7, la version 3.0 a apporté des modifications profondes. Il faut noter que Python 2.7 prendra “[sa retraite](#)” le premier janvier 2020. Passée cette date, le support ne sera plus assuré.

## 1.3 Espace de travail

Il existe de nombreux environnements dans lesquels programmer en Python. Nous allons en présenter succinctement quelques uns.

Il est supposé ici que vous vous avez installé [Anaconda](#) sur votre poste. Anaconda est une distribution gratuite et open source des langages de programmation Python et R pour les applications en *data science* et apprentissage automatique. Par ailleurs, lorsqu’il est fait mention du terminal dans les notes, il est supposé que le système d’exploitation de votre machine est soit Linux, soit Mac OS.

### 1.3.1 Python dans un terminal

Il est possible d’appeler Python depuis un terminal, en exécutant la commande suivante (sous Windows : dans le menu démarrer, lancer le logiciel “Python 3.6”) :

```
python
```

Ce qui donne le rendu visible sur la Figure 1.2 :



```
iMac-de-Ewen:~ ewengallic$ python
Python 3.6.5 [Anaconda, Inc.] (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

FIGURE 1.2 – Python dans un terminal.

On note la présence des caractères `>>>` (*prompt*), qui invitent l'utilisateur à inscrire une commande. Les expressions sont évaluées une fois qu'elle sont soumises (à l'aide de la touche `ENTREE`) et le résultat est donné, lorsqu'il n'y a pas d'erreur dans le code.

Par exemple, lorsque l'on évalue `2+1` :

```
>>> 2+1
3
>>>
```

On note la présence du *prompt* à la fin, indiquant que Python est prêt à recevoir de nouvelles instructions.

### 1.3.2 IPython

Il existe un environnement un peu plus chaleureux que Python dans le terminal : IPython. Il s'agit également d'un terminal interactif, mais avec davantage de fonctionnalités, notamment la coloration syntaxique ou l'auto-complétion (en utilisant la touche de tabulation).

Pour lancer IPython, on peut ouvrir un terminal et taper (puis valider) :

```
ipython
```

On peut également lancer IPython depuis la fenêtre d'accueil d'Anaconda, en cliquant sur le bouton **Launch** de l'application `qtconsole`, visible sur la Figure [1.3](#).

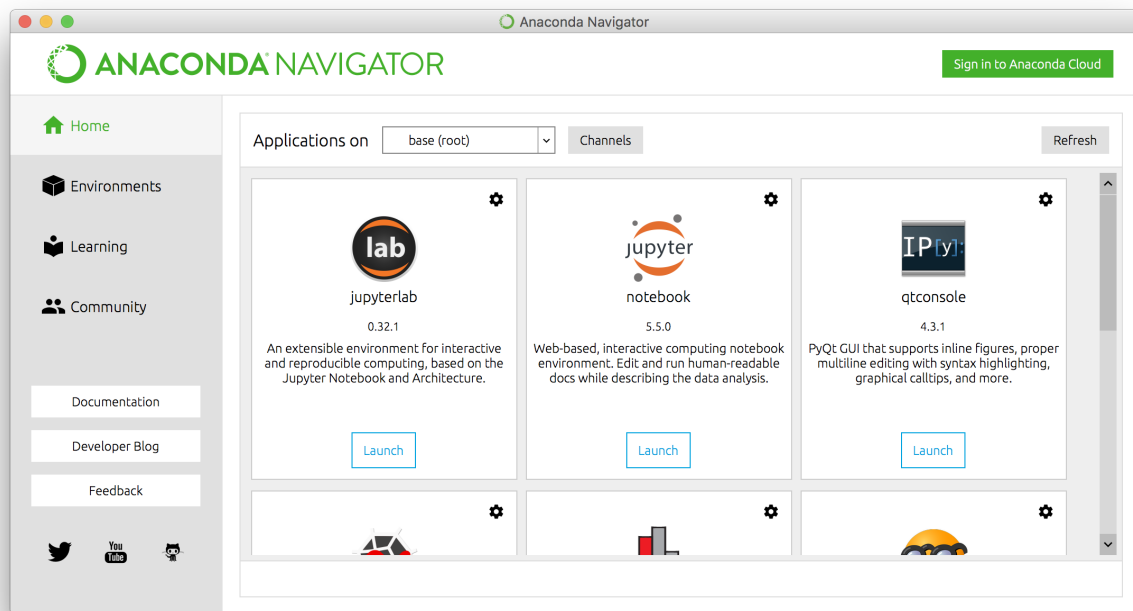


FIGURE 1.3 – Fenêtre d'accueil d'Anaconda.

La console IPython, une fois lancée, ressemble à ceci :

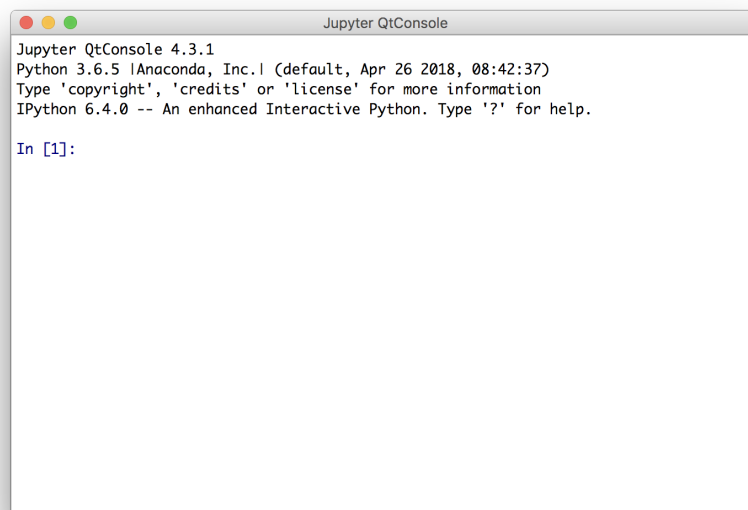


FIGURE 1.4 – Console IPython.

Soumettons une instruction simple pour évaluation à Python :

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

Le résultat donne :

```
In [1]: print("Hello World")
Hello World

In [2]:
```

Plusieurs choses sont à noter. Premièrement, on note qu'à la fin de l'exécution de l'instruction, IPython nous indique qu'il est prêt à recevoir de nouvelles instruction, par la présence du *prompt* `In [2]:`. Le numéro entre les crochets désigne le numéro de l'instruction. On note qu'il est passé de 1 à 2 après l'exécution. Ensuite, on note que le résultat de l'appel à la fonction `print()`, avec la chaîne de caractères (délimitée par des guillemets), affiche à l'écran ce qui était contenu entre les parenthèses.

### 1.3.3 Spyder

Tandis que lorsqu'on utilise Python via un terminal, il est préférable d'avoir un éditeur de texte ouvert à côté (pour pouvoir sauvegarder les instructions), comme, par exemple, [Sublime Text](#) sous Linux ou Mac OS, ou [notepad++](#) sous Windows.

Une autre alternative consiste à utiliser un environnement de développement (IDE, pour *Integrated development environment*) unique proposant notamment, à la fois un éditeur et une console. C'est ce que propose [Spyder](#), avec en outre de nombreuses fonctionnalités supplémentaires, comme la gestion de projet, un explorateur de fichier, un historique des commandes, un débbugger, etc.

Pour lancer Spyder, on peut passer par un terminal, en évaluant tout simplement **Spyder** (ou en lançant le logiciel depuis le menu démarrer sous Windows). Il est également possible de lancer Spyder depuis Anaconda.

L'environnement de développement, comme visible sur la Figure 1.5, se décompose en plusieurs fenêtres :

- à gauche : l'éditeur de script ;
- en haut à droite : une fenêtre permettant d'afficher l'aide de Python, l'arborescence du système ou encore les variables créées ;
- en bas à droite : une ou plusieurs consoles.



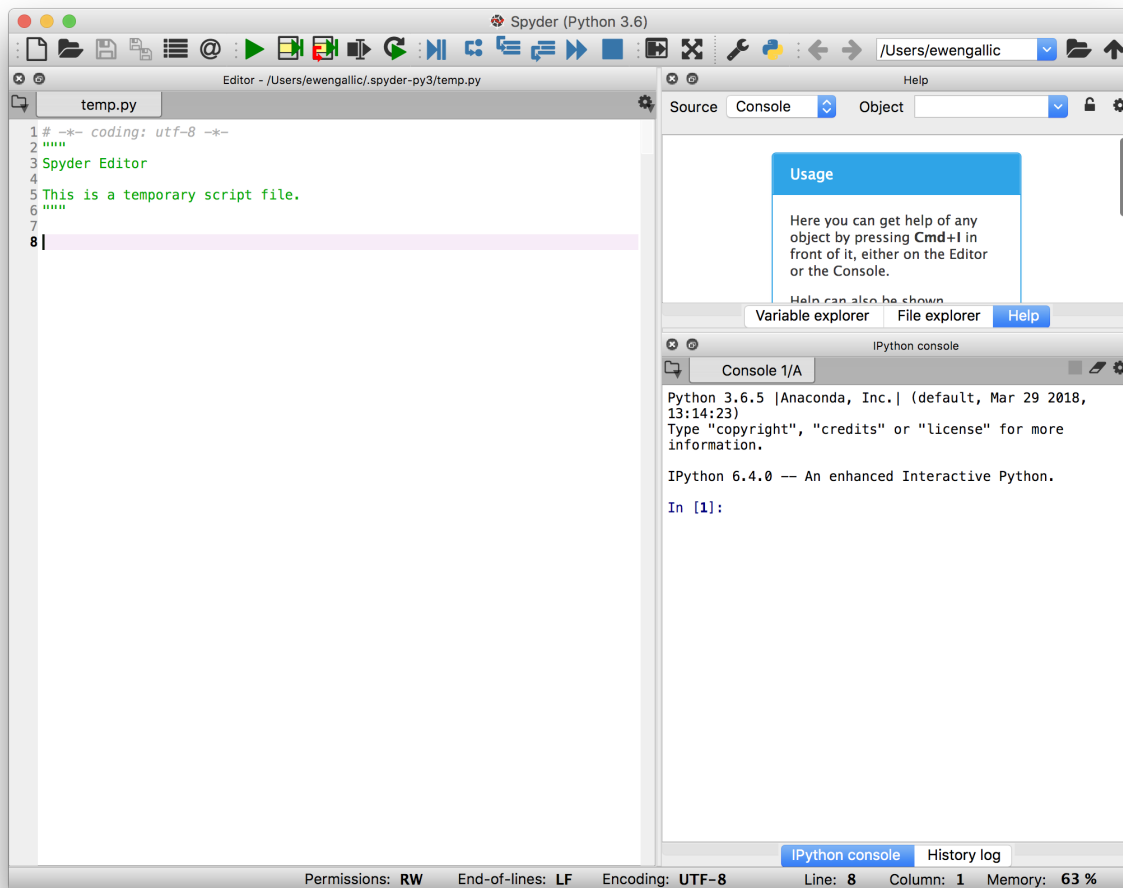


FIGURE 1.5 – Spyder.

### 1.3.4 Jupyter

Il existe une interface graphique par navigateur d'IPython, appelée [Jupyter Notebook](#). Il s'agit d'une application en open-source permettant de créer et partager des documents qui contiennent du code, des équations, des représentations graphiques et du texte. Il est possible de faire figurer et exécuter des codes de langages différents dans les notebook Jupyter.

Pour lancer Jupyter, on peut passer par Anaconda. Après avoir cliqué sur le bouton **Launch**, de Jupyter Notebook, le navigateur web se lance et propose une arborescence, comme montré sur la Figure 1.6. Sans que l'on s'en rendiez compte, un serveur local web a été lancé ainsi qu'un processus Python (un *kernel*).

Si le navigateur ne se lance pas automatiquement, on peut accéder à la page qui aurait dû s'afficher, en se rendant à l'adresse suivante : <http://localhost:8890/tree?>.

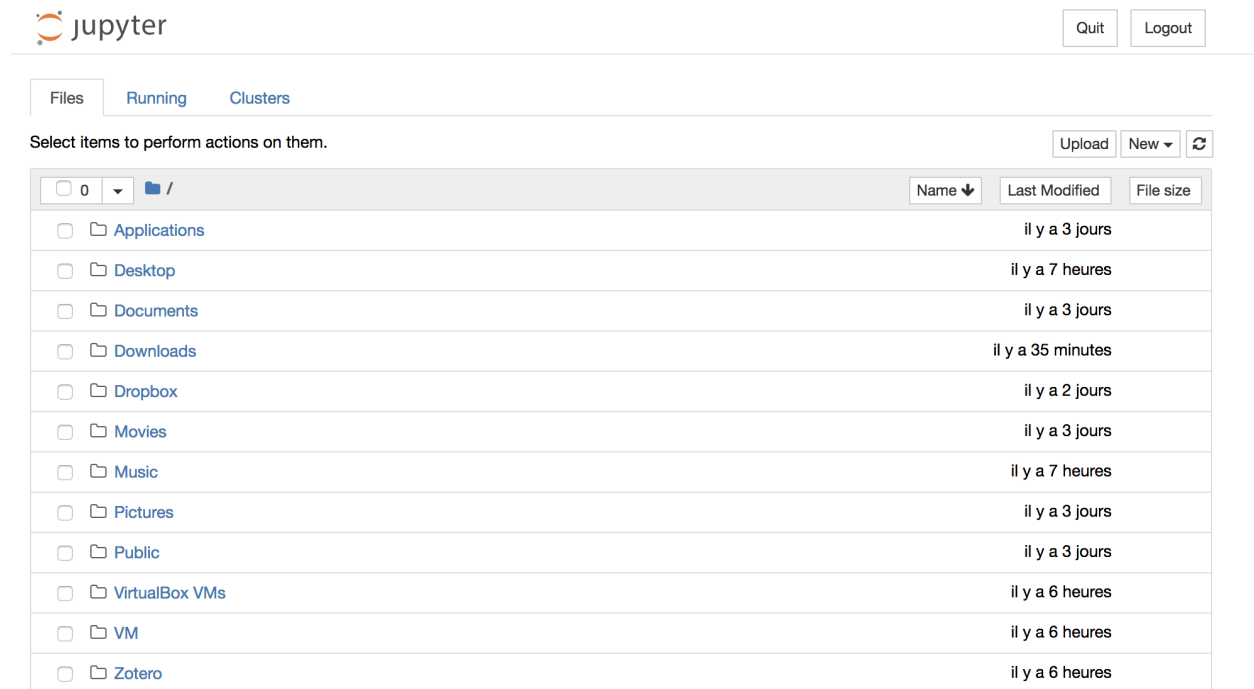


FIGURE 1.6 – Jupyter.

Pour aborder les principales fonctions de Jupyter, nous allons créer un dossier `jupyter` dans un répertoire de notre choix. Une fois ce dossier créé, y naviguer à travers l’arborescence de Jupyter, dans le navigateur web.

Une fois dans le dossier, créer un nouveau Notebook `Python 3` (en cliquant sur le bouton **New** en haut à gauche de la fenêtre, puis sur `Python 3`).

Un notebook intitulé `Untitled` vient d’être créé, la page affiche un document vide, comme visible sur la Figure 1.7.

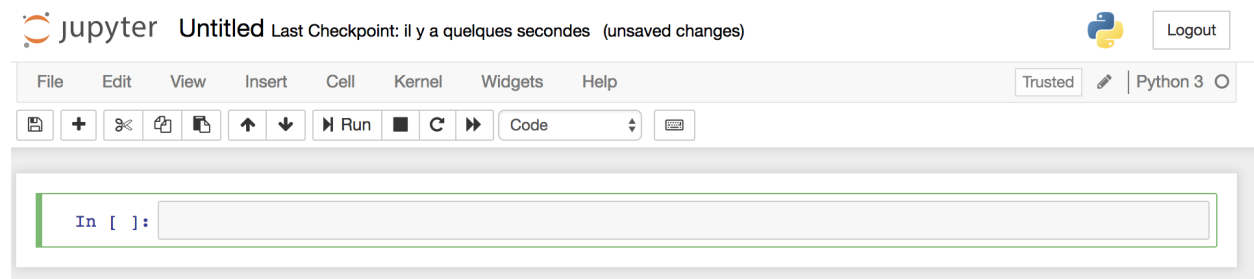


FIGURE 1.7 – Un notebook vide.

Si on regarde dans notre explorateur de fichier, dans le dossier `jupyter` fraîchement créé, un nouveau fichier est apparu : `Untitled.ipynb`.

#### 1.3.4.1 Évaluation d'une instruction

Retournons dans le navigateur web, sur la page affichant notre *notebook*.

En dessous de la barre des menus, on note la présence d'une zone encadrée, **une cellule**, commençant, à l'instar de ce que l'on voyait dans la console sur IPython, par `IN []:`. À droite, la zone grisée nous invite à soumettre des instructions en Python.

Inscrivons :

```
2+1
```

Pour soumettre l'instruction à évaluation, il existe plusieurs manières (il s'assurer d'avoir cliqué à l'intérieur de la cellule) :

- dans la barre des menus : `Cell > Run Cells`;
- dans la barre des raccourcis : bouton `Run`;
- avec le clavier : maintenir la touche `CTRL` et presser sur `Entree`.

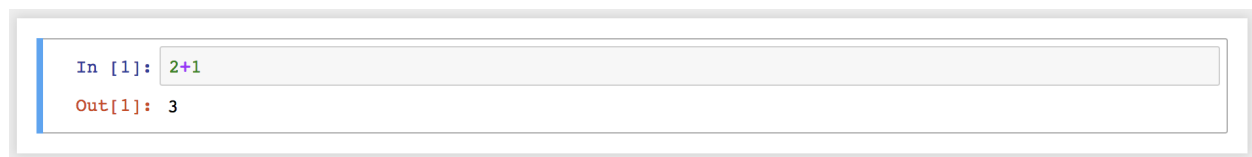


FIGURE 1.8 – Cellule évaluée.

#### 1.3.4.2 Cellules de texte

Un des intérêts des *notebooks* est qu'il est possible d'ajouter des cellules de texte.

Ajoutons une cellule en-dessous de la première. Pour ce faire, on peut procéder soit :

- par la barre de menu : `Insert > Insert Cell Below` (pour insérer une cellule en-dessous ; si on désire une insertion au-dessus, il suffit de choisir `Insert Cell Above`) ;
- en cliquant dans le cadre de la cellule à partir de laquelle on désire faire un ajout (n'importe où, sauf dans la zone grisée de code, de manière à passer en mode **commande**), puis en appuyant sur la touche `B` du clavier (`A` pour une insertion au-dessus).

La nouvelle cellule appelle à nouveau à inscrire une instruction en Python. Pour indiquer que le contenu doit être interprété comme du texte, il est nécessaire de le préciser. Encore une fois, plusieurs méthodes permettent de le faire :

- par la barre de menu : `Cell > Cell Type > Markdown` ;
- par la barre des raccourcis : dans le menu déroulant où est inscrit `Code`, en sélectionnant `Markdown` ;
- en mode commande (après avoir cliqué à l'intérieur du cadre de la cellule, mais pas dans la zone de code), en appuyant sur la touche `M` du clavier.

La cellule est alors prête à recevoir du texte, rédigé en markdown. Pour plus d'informations sur la rédaction en Markdown, se référer à cette [antisèche](#) par exemple.

Entrons quelques lignes de texte pour voir très rapidement le fonctionnement des cellules rédigées en Markdown.

```
# Un titre de niveau 1
```

```
Je vais écrire *du texte en italique* et aussi **en gras**.
```

```
## Un titre de niveau 2
```

```
Je peux faire des listes :
```

- avec un item ;
- un second ;
- et un troisième imbriquant une nouvelle liste :
  - avec un sous-item,
  - et un second ;
- un quatrième incluant une liste imbriquée numérotée :
  1. avec un sous-item,
  1. et un autre.

```
## Un autre titre de niveau 2
```

```
Je peux même faire figurer des équation  $\LaTeX$ .
```

```
Comme par exemple  $X \sim \mathcal{N}(0,1)$ .
```

```
Pour en savoir plus sur  $\LaTeX$ , on peut se référer à cette :  
[page Wikipédia](https://en.wikibooks.org/wiki/LaTeX/Mathematics).
```

Ce qui donne, dans Jupyter :

Reste alors à l'évaluer, comme s'il s'agissait d'une cellule contenant une instruction Python, pour basculer vers un affichage Markdown (CTRL et ENTREE).

Pour **éditer le texte** une fois que l'on a basculé en markdown, un simple double-clic dans la zone de texte de la cellule fait l'affaire.

Pour **changer le type de la cellule pour qu'elle devienne du code** :

- par la barre de menu : Cell > Cell Type > Code ;
- par la barre des raccourcis : dans le menu déroulant où est inscrit Code, en sélectionnant Code ;
- en mode commande, appuyer sur la touche du clavier Y.

```
In [1]: 2+1
```

```
Out[1]: 3
```

## # Un titre de niveau 1

Je vais écrire *\*du texte en italique\** et aussi **\*\*en gras\*\***.

## ## Un titre de niveau 2

Je peux faire des listes :

- avec un item ;
- un second ;
- et un troisième imbriquant une nouvelle liste :
  - avec un sous-item,
  - et un second ;
- un quatrième incluant une liste imbriquée numérotée :
  1. avec un sous-item,
  1. et un autre.

## ## Un autre titre de niveau 2

Je peux même faire figurer des équation  $\LaTeX$ , comme par exemple  $\sim \mathcal{N}(0,1)$ .

Pour en savoir plus sur  $\LaTeX$ , on peut se référer à cette [page Wikipédia](https://en.wikibooks.org/wiki/LaTeX/Mathematics) (<https://en.wikibooks.org/wiki/LaTeX/Mathematics>).

FIGURE 1.9 – Cellule textuelle non évaluée.

### 1.3.4.3 Suppression d’une cellule

Pour supprimer une cellule :

- par la barre de menu : Edit > Delete Cells;
- par la barre des raccourcis : icône en forme de ciseaux;
- en mode commande, appuyer deux fois sur la touche du clavier D.

## 1.4 Les variables

### 1.4.1 Assignment et suppression

Lorsque nous avons évalué les instructions `2+1` précédemment, le résultat s’est affiché dans la console, mais il n’a pas été enregistré. Dans de nombreux cas, il est utile de conserver le contenu du résultat dans un objet, pour pouvoir le réutiliser par la suite. Pour ce faire, on utilise des *variables*. Pour créer une variable, on utilise le signe d’égalité (=), que l’on fait suivre par ce que l’on veut sauvegarder (du texte, un nombre, plusieurs nombres, etc.) et précéder par le nom que l’on utilisera pour désigner cette variable.

Par exemple, si on souhaite stocker le résultat du calcul `2+1` dans une variable que l’on nommera `x`, il faudra écrire :

```
x = 2+1
```

Pour afficher la valeur de notre variable `x`, on fait appel à la fonction `print()` :

```
print(x)
```

```
## 3
```

Pour changer la valeur de la variable, il suffit de faire une nouvelle assignation :

```
x = 4  
print(x)
```

```
## 4
```

Il est également possible de donner plus d'un nom à un même contenu (on réalise une copie de `x`) :

```
x = 4;  
y = x;  
print(y)
```

```
## 4
```

Si on modifie la copie, l'original ne sera pas affecté :

```
y = 0  
print(y)
```

```
## 0
```

```
print(x)
```

```
## 4
```

Pour **supprimer** une variable, on utilise l'instruction `del` :

```
del y
```

L'affichage du contenu de `y` renvoie une erreur :

```
print(y)
```

```
## NameError: name 'y' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Mais on note que la variable `x` n'a pas été supprimée :

```
print(x)
```

```
## 4
```

### 1.4.2 Conventions de nommage

Le nom d'une variable peut être composé de caractères alphanumériques ainsi que du trait de soulignement (`_`) (il n'y a pas de limite sur la longueur du nom). Il est proscrit de faire commencer le nom de la variable par un nombre. Il est également interdit de faire figurer une espace dans le nom d'une variable.

Pour accroître la lisibilité du nom des variables, plusieurs méthodes existent. Nous adopterons la suivante :

- toutes les lettres en minuscule ;
- la séparation des termes par un trait de soulignement.

Exemple, pour une variable contenant la valeur de l'identifiant d'un utilisateur : `id_utilisateur`.

Il faut noter que le nom des variables est **sensible à la casse** :

```
x = "toto"
print(x)
```

```
## toto
```

```
print(X)
```

```
## NameError: name 'X' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

## 1.5 Les commentaires

Pour ajouter des commentaires en python, il existe plusieurs façons.

Une des manières de faire est d'utiliser le symbole dièse (#) pour effectuer un **commentaire sur une seule ligne**. Tout ce qui suit le dièse jusqu'à la fin de la ligne ne sera pas évalué par Python. En revanche, ce qui vient avant le dièse le sera.

```
# Un commentaire print("Bonjour")  
print("Hello") # Un autre commentaire
```

```
## Hello
```

L'introduction d'un **bloc de commentaires** (des commentaires sur plusieurs lignes) s'effectue quant à elle en entourant ce qui est ) commenter d'un délimiteur : trois guillemets simples ou doubles :

```
"""  
Un commentaire qui commencer sur une ligne  
et qui continue sur une autre  
et s'arrête à la troisième  
"""
```

## 1.6 Les modules et les packages

Certaines fonctions de base en Python sont chargées par défaut. D'autres, nécessitent de charger un **module**. Ces modules sont des fichiers qui contiennent des **définitions** ainsi que des **instructions**.

Lorsque plusieurs modules sont réunis pour offrir un ensemble de fonctions, on parle alors de **package**.

Parmi les *packages* qui seront utilisés dans ces notes, on peut citer :

- [NumPy](#), un *package* fondamental pour effectuer des calculs scientifiques ;
- [pandas](#), un *package* permettant de manipuler facilement les données et de les analyser ;
- [Matplotlib](#), un *package* permettant de réaliser des graphiques.

Pour charger un module (ou un *package*), on utilise la commande **import**. Par exemple, pour charger le *package* **pandas** :

```
import pandas
```

Ce qui permet de faire appel à des fonctions contenues dans le module ou le *package*. Par exemple, ici, on peut faire appel à la fonction **Series()**, contenue dans le *package* **pandas**, permettant de créer un tableau de données indexées à une dimension :



```
x = pandas.Series([1, 5, 4])
print(x)
```

```
## 0      1
## 1      5
## 2      4
## dtype: int64
```

Il est possible de donner un alias au module ou au *package* que l'on importe, en le précisant à l'aide de la syntaxe suivante :

```
import module as alias
```

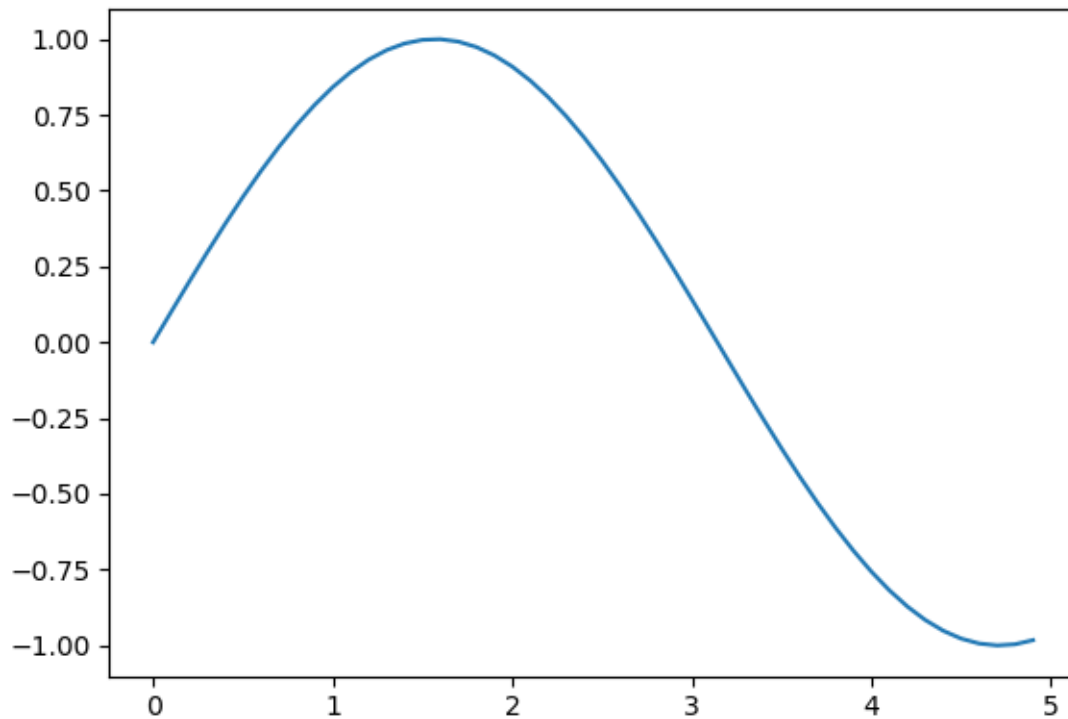
Cette pratique est courante pour abréger les noms des modules que l'on va être amené à utiliser beaucoup. Par exemple, pour `pandas`, il est coutume d'écourter le nom en `pd` :

```
import pandas as pd
x = pd.Series([1, 5, 4])
print(x)
```

```
## 0      1
## 1      5
## 2      4
## dtype: int64
```

On peut également importer une seule fonction d'un module, et lui attribuer (optionnellement) un alias. Par exemple, avec la fonction `pyplot` du *package* `matplotlib`, il est coutume de faire comme suit :

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 5, 0.1);
y = np.sin(x)
plt.plot(x, y)
```



## 1.7 L'aide

Pour conclure cette introduction, il semble important de mentionner la présence de l'**aide** et de la **documentation** en Python.

Pour obtenir des informations sur des fonctions, il est possible de se référer à la [documentation en ligne](#). Il est également possible d'obtenir de l'aide à l'intérieur de l'environnement que l'on utilise, en utilisant le point d'interrogation (?).

Par exemple, lorsque l'on utilise IPython (ce qui, rappelons-le, est le cas dans Jupyter), on peut accéder à l'aide à travers différentes syntaxes :

- ? : fournit une introduction et un aperçu des fonctionnalités offertes en Python (on la quitte avec la touche **ESC** par exemple);
- `object?` : fournit des détails au sujet de 'object' (par exemple `x?` ou encore `plt.plot?`);
- `object??` : plus de détails à propos de 'object';
- `%quickref` : référence courte sur les syntaxes en Python;
- `help()` : accès à l'aide de Python.

*Note* : la touche de **tabulation** du clavier permet non seulement une **autocomplétion**, mais

aussi une **exploration du contenu** d'un objet ou module.

Par ailleurs, lorsqu'il s'agit de trouver de l'aide sur un problème plus complexe, le bon réflexe à adopter est de ne pas hésiter à chercher sur un moteur de recherche, dans des mailing-lists et bien évidemment sur les nombreuses questions sur [Stack Overflow](#).



# Chapitre 2

## Types de données

Il existe quelques types de données intégrés dans Python. Nous allons dans cette partie évoquer les chaînes de caractères, les valeurs numériques, les booléens (TRUE/FALSE), la valeur `null` et les dates et temps.

### 2.1 Chaînes de caractères

Une chaîne de caractères, ou *string* en anglais, est une collection de caractères comme des lettres, des nombres, des espaces, des signes de ponctuation, etc.

Les chaînes de caractères sont repérées à l'aide de guillemets simples, doubles, ou triples.

Voici un exemple :

```
x = "Hello World"
```

Pour afficher dans la console le contenu de notre variable `x` contenant la chaîne de caractères, on fait appel à la fonction `print()` :

```
print(x)
```

```
## Hello World
```

Comme indiqué juste avant, des guillemets simples peuvent être utilisés pour créer une chaîne de caractères :

```
y = 'How are you?'  
print(y)
```

```
## How are you?
```

Pour faire figurer des apostrophes dans une chaîne de caractères créée à l'aide de guillemets simples, il est nécessaire d'utiliser un caractère d'échappement : une barre oblique inversée (`\`) :

```
z = 'I\'m fine'
print(z)
```

```
## I'm fine
```

On peut noter que si la chaîne de caractères est créée à l'aide de guillemets doubles, il n'est pas nécessaire d'avoir recours au caractère d'échappement :

```
z = "I'm \"fine\""
print(z)
```

```
## I'm "fine"
```

Pour indiquer un retour à la ligne, on utilise la chaîne `\n` :

```
x = "Hello, \nWorld"
print(x)
```

```
## Hello ,
## World
```

Dans le cas de chaînes de caractères sur **plusieurs lignes**, le fait d'utiliser des guillemets simples ou doubles renverra une erreur (*EOL while scanning trial literal, i.e.*, détection d'une erreur de syntaxe, Python s'attendait à quelque chose d'autre à la fin de la ligne). Pour écrire une chaîne de caractères sur plusieurs lignes, Python propose d'utiliser trois fois des guillemets (simples ou doubles) en début et fin de chaîne :

```
x = """Hello,
World"""
print(x)
```

```
## Hello ,
## World
```

## Remarque 2.1.1

Le caractère `\` (barre oblique inversée, ou *backslash*) est le caractère d'échappement. Il permet d'afficher certains caractères, comme les guillemets dans une chaîne elle-même définie à l'aide de guillemets, ou bien les caractères de contrôle, comme la tabulation, le saut de ligne, etc. Voici quelques exemples courants :

Code	Description	Code	Description
<code>\n</code>	Nouvelle ligne	<code>\r</code>	Retour à la ligne
<code>\t</code>	Tabulation	<code>\b</code>	Retour arrière
<code>\</code>	Barre oblique inversée	<code>\'</code>	Apostrophe
<code>\"</code>	Apostrophe double	<code>\`</code>	Accent grave

Pour récupérer la **longueur d'une chaîne de caractères**, Python propose la fonction `len()` :

```
x = "Hello World !"
print(len(x))
```

```
## 13
```

```
print(x, len(x))
```

```
## Hello World ! 13
```

### 2.1.1 Concaténation de chaînes

Pour concaténer des chaînes de caractères, c'est-à-dire les mettre bout à bout, Python propose d'utiliser l'opérateur `+` :

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

```
## Hello World
```

L'opérateur `*` permet quant à lui de répéter plusieurs fois une chaîne :

```
print( 3 * "Go Habs Go! " + "Woo Hoo!")
```

```
## Go Habs Go! Go Habs Go! Go Habs Go! Woo Hoo!
```

Lorsque deux littéraux de chaînes sont côte à côte, Python les concatène :

```
x = ('You shall ' 'not ' "pass!")
print(x)
```

```
## You shall not pass!
```

Il est également possible d'ajouter à une chaîne de caractères le contenu d'une variable, à l'aide du marqueur %s :

```
x = "J'aime coder en %s"
langage_1 = "R"
langage_2 = "Python"
preference_1 = x % langage_1
print(preference_1)
```

```
## J'aime coder en R
```

```
preference_2 = x % langage_2
print(preference_2)
```

```
## J'aime coder en Python
```

Il est tout à fait possible d'ajouter **plus d'un contenu de variable** dans une chaîne de caractères, toujours avec le marqueur %s :

```
x = "J'aime coder en %s et en %s"
preference_3 = x % (langage_1, langage_2)
print(preference_3)
```

```
## J'aime coder en R et en Python
```

## 2.1.2 Indexation et extraction

Les chaînes de caractères peuvent être indexées. Attention, **\*\*l'indice du premier caractère commence à 0\***.

Pour obtenir le *i*e caractère d'une chaîne, on utilise des crochets. La syntaxe est la suivante :

```
x[i-1]
```

Par exemple, pour afficher le premier caractère, puis le cinquième de la chaîne `Hello` :



```
x = "Hello"  
print(x[0])
```

```
## H
```

```
print(x[4])
```

```
## o
```

L'extraction peut s'effectuer en partant par la fin de la chaîne, en faisant précéder la valeur de l'indice par le signe moins (-).

Par exemple, pour afficher l'avant-dernier caractère de notre chaîne `x` :

```
print(x[-2])
```

```
## l
```

L'extraction d'une sous-chaîne en précisant sa position de début et de fin (implicitement ou non) s'effectue avec les crochets également. Il suffit de préciser les deux valeurs d'indices : `[debut:fin]`.

```
x = "You shall not pass!"  
# Du quatrième caractère (non inclus) au neuvième (inclus)  
print(x[4:9])
```

```
## shall
```

Lorsque l'on ne précise pas la première valeur, le début de la chaîne est pris par défaut ; lorsque le second n'est pas précisé, la fin de la chaîne est prise par défaut.

```
# Du 4e caractère (non inclus) à la fin de la chaîne  
print(x[4:])  
# Du début de la chaîne à l'avant dernier caractère (inclus)  
print(x[:-1])  
# Du 3e caractère avant la fin (inclus) jusqu'à la fin  
print(x[-5:])
```

```
## shall not pass!
```

```
## You shall not pass
```

```
## pass!
```

Il est possible de rajouter un troisième indice dans les crochets : **le pas**.

```
# Du 4e caractère (non inclus), jusqu'à la fin de la chaîne,  
# par pas de 3.  
print(x[4::3])
```

```
## sln s
```

Pour obtenir la chaîne en dans le sens opposé :

```
print(x[::-1])
```

```
## !ssap ton llaHS uoY
```

### 2.1.3 Méthodes disponibles avec les chaînes de caractères

De nombreuses méthodes sont disponibles pour les chaînes de caractères. En ajoutant un point (.) après le nom d'un objet désignant une chaîne de caractères puis en appuyant sur la touche de tabulation, les méthodes disponibles s'affichent dans un menu déroulant.

Par exemple, la méthode `count()` permet de compter le nombre d'occurrences d'un motif dans la chaîne. Pour compter le nombre d'occurrence de `in` dans la chaîne suivante :

```
x = "le train de tes injures roule sur le rail de mon indifférence"  
print(x.count("in"))
```

```
## 3
```

#### Remarque 2.1.2

Une fois l'appel à méthode écrit, en plaçant le curseur à la fin de la ligne et en appuyant sur les touches **Shift** et **Tabulation**, on peut afficher des explications.

#### 2.1.3.1 Conversion en majuscules ou en minuscules

Les méthodes `lower()` et `upper()` permettent de passer une chaîne de caractères en caractères minuscules et majuscules, respectivement.

```
x = "le train de tes injures roule sur le rail de mon indifférence"
print(x.lower())
print(x.upper())

## le train de tes injures roule sur le rail de mon indifférence

## LE TRAIN DE TES INJURES ROULE SUR LE RAIL DE MON INDIFFÉRENCE
```

### 2.1.3.2 Recherche de chaînes de caractères

Quand on souhaite **retrouver un motif** dans une chaîne de caractères, on peut utiliser la méthode `find()`. On fournit en paramètres un motif à rechercher. La méthode `find()` retourne le plus petit indice dans la chaîne où le motif est trouvé. Si le motif n'est pas retrouvé, la valeur retournée est `-1`.

```
print(x.find("in"))
print(x.find("bonjour"))
```

```
## 6
```

```
## -1
```

Il est possible d'ajouter en option une indication permettant de **limiter la recherche sur une sous-chaîne**, en précisant l'indice de début et de fin :

```
print(x.find("in", 7, 20))
```

```
## 16
```

Note : on peut omettre l'indice de fin ; en ce cas, la fin de la chaîne est utilisée :

```
print(x.find("in", 20))
```

```
## 49
```

#### Remarque 2.1.3

Si on ne désire pas connaître la position de la sous-chaîne, mais uniquement sa présence ou son absence, on peut utiliser l'opérateur `in` : `print("train" in x)`

Pour effectuer une recherche **sans prêter attention à la casse**, on peut utiliser la méthode `capitalize()` :

```
x = "Mademoiselle Deray, il est interdit de manger de la choucroute ici."
print(x.find("deray"))
```

```
## -1
```

```
print(x.capitalize().find("deray"))
```

```
## 13
```

### 2.1.3.3 Découpage en sous-chaînes

Pour **découper une chaîne de caractères en sous-chaînes**, en fonction d'un motif servant à la délimitation des sous-chaînes (par exemple une virgule, ou une espace), on utilise la méthode `split()` :

```
print(x.split(" "))
```

```
## ['Mademoiselle', 'Deray,', 'il', 'est', 'interdit', 'de', 'manger',
    ', 'de', 'la', 'choucroute', 'ici.']
```

En indiquant en paramètres une valeur numérique, on peut limiter le nombre de sous-chaînes retournées :

```
# Le nombre de sous-chaînes maximum sera de 3
```

```
print(x.split(" ", 3))
```

```
## ['Mademoiselle', 'Deray,', 'il', 'est interdit de manger de la',
    'choucroute ici.']
```

La méthode `splitlines()` permet également de séparer une chaîne de caractères en fonction d'un motif, ce motif étant un caractère de fin de ligne, comme un saut de ligne ou un retour chariot par exemple.

```
x = '''Luke, je suis ton pere !
- Non... ce n'est pas vrai ! C'est impossible !
- Lis dans ton coeur, tu sauras que c'est vrai.
- Nooooooooon ! Nooooon !'''
print(x.splitlines())
```

```
## ["Luke, je suis ton pere !", "- Non... ce n'est pas vrai ! C'est impossible !", "- Lis dans ton coeur, tu sauras que c'est vrai .", '- Nooooooooon ! Nooooon !"]
```

### 2.1.3.4 Nettoyage, complétion

Pour retirer des caractères blancs (*e.g.*, des espaces, sauts de ligne, quadratins, etc.) présents en début et fin de chaîne, on peut utiliser la méthode `strip()`, ce qui est parfois très utile pour nettoyer des chaînes.

```
x = "\n\n    Pardon, du sucre ?    \n \n"
print(x.strip())
```

```
## Pardon, du sucre ?
```

On peut préciser en paramètre quels caractères retirer en début et fin de chaîne :

```
x = "www.egallic.fr"
print(x.strip("wrf."))
```

```
## egallic
```

Parfois, il est nécessaire de s'assurer d'obtenir une **chaîne d'une longueur donnée** (lorsque l'on doit fournir un fichier avec des largeurs fixes pour chaque colonne par exemple). La méthode `rjust()` est alors d'un grand secours. En lui renseignant une longueur de chaîne et un caractère de remplissage, elle retourne la chaîne de caractères avec une complétion éventuelle (si la longueur de la chaîne retournée n'est pas assez longue au regard de la valeur demandée), en répétant le caractère de remplissage autant de fois que nécessaire.

Par exemple, pour avoir une coordonnée de longitude, stockée dans une chaîne de caractères de longueur 7, en rajoutant des espaces si nécessaire :

```
longitude = "48.11"
print(x.rjust(7, " "))
```

```
## www.egallic.fr
```

### 2.1.3.5 Remplacements

La méthode `replace()` permet d'effectuer des **remplacements de motifs** dans une chaîne de caractères.

```
x = "Criquette ! Vous, ici ? Dans votre propre salle de bain ? Quelle surprise !"
print(x.replace("Criquette", "Ridge"))
```

```
## Ridge ! Vous, ici ? Dans votre propre salle de bain ? Quelle
    surprise !
```

Cette méthode est très pratique pour **retirer des espaces** par exemple :

```
print(x.replace(" ", ""))
```

```
## Criquette!Vous,ici?Dansvotrepropresalledebain?Quellesurprise!
```

Voici un tableau répertoriant quelques méthodes disponibles ([liste exhaustive dans la documentation](#)) :

Méthode	Description
<code>capitalize()</code>	Mise en majuscule du premier caractère et en minuscule du reste
<code>casefold()</code>	retire les distinctions de casse (utile pour la comparaison de chaînes sans faire attention à la casse)
<code>count()</code>	Compte le nombre d'occurrence (sans chevauchement) d'un motif
<code>encode()</code>	Encode une chaîne de caractères dans un encodage spécifique
<code>find()</code>	Retourne le plus petit indice où une sous-chaîne est trouvée
<code>lower()</code>	Retourne la chaîne en ayant passé chaque caractère alphabétique en minuscules
<code>replace()</code>	Remplace un motif par un autre
<code>split()</code>	Sépare la chaîne en sous-chaînes en fonction d'un motif
<code>title()</code>	Retourne la chaîne en ayant passé chaque première lettre de mot par une majuscule
<code>upper()</code>	Retourne la chaîne en ayant passé chaque caractère alphabétique en majuscules

### 2.1.4 Conversion en chaînes de caractères

Lorsque l'on veut concaténer une chaîne de caractères avec un nombre, Python retourne une erreur.

```
nb_followers = 0
message = "He has " + nb_followers + "followers."
```

```
## TypeError: must be str, not int
##
## Detailed traceback:
```

```
## File "<string>", line 1, in <module>
```

```
print(message)
```

```
## NameError: name 'message' is not defined
##
## Detailed traceback:
## File "<string>", line 1, in <module>
```

Il est alors nécessaire de convertir au préalable l'objet n'étant pas une chaîne en une chaîne de caractères. Pour ce faire, Python propose la fonction `str()` :

```
message = "He has " + str(nb_followers) + " followers."
print(message)
```

```
## He has 0 followers.
```

### 2.1.5 Exercice

1. Créer deux variables nommées `a` et `b` afin qu'elles contiennent respectivement les chaînes de caractères suivantes : 23 à 0 et C'est la piquette, Jack!.
2. Afficher le nombre de caractères de `a`, puis de `b`.
3. Concaténer `a` et `b` dans une seule chaîne de caractères, en ajoutant une virgule comme caractère de séparation.
4. Même question en choisissant une séparation permettant un retour à la ligne entre les deux phrases.
5. À l'aide de la méthode appropriée, mettre en majuscules `a` et `b`.
6. À l'aide de la méthode appropriée, mettre en minuscules `a` et `b`.
7. Extraire le mot `la` et `Jack` de la chaîne `b`, en utilisant les indices.
8. Rechercher si la sous-chaîne `piqu` est présente dans `b`, puis faire de même avec la sous-chaîne `mauvais`.
9. Retourner la position (indice) du premier caractère `a` retrouvé dans la chaîne `b`, puis essayer avec le caractère `w`.
10. Remplacer les occurrences du motif `a` par le motif `Z` dans la sous-chaîne `b`.
11. Séparer la chaîne `b` en utilisant la virgule comme séparateur de sous-chaînes.
12. (Bonus) Retirer tous les caractères de ponctuation de la chaîne `b`, puis utiliser une méthode appropriée pour retirer les caractères blancs en début et fin de chaîne. (Utiliser la librairie `regex`).

## 2.2 Valeurs numériques

Il existe quatre catégories de nombres en Python : les entiers, les nombres à virgule flottante et les complexes.

### 2.2.1 Entiers

Les entiers (`ints`), en Python, sont des nombres entiers signés.

#### Remarque 2.2.1

On accède au type d'un objet à l'aide de la fonction `type()` en Python.

```
x = 2
y = -2
print(type(x))

## <class 'int'>
```

```
print(type(y))

## <class 'int'>
```

### 2.2.2 Nombre à virgule flottante

Les nombres à virgule flottante (`floats`) représentent les nombres réels. Ils sont écrits à l'aide d'un point permettant de distinguer la partie entière de la partie décimale du nombre.

```
x = 2.0
y = 48.15162342
print(type(x))

## <class 'float'>
```

```
print(type(y))

## <class 'float'>
```

Il est également possible d'avoir recours aux notations scientifiques, en utilisant `E` ou `e` pour indiquer une puissance de 10. Par exemple, pour écrire  $3,2^{12}$ , on procèdera comme suit :



```
x = 3.2E12
y = 3.2e12
print(x)
```

```
## 3200000000000.0
```

```
print(y)
```

```
## 3200000000000.0
```

### 2.2.3 Nombres complexes

Python permet nativement de manipuler des nombres complexes, de la forme  $z = a + ib$ , où  $a$  et  $b$  sont des nombres à virgule flottante, et tel que  $i^2 = (-i)^2 = -1$ . La partie réelle du nombre,  $\Re(z)$ , est  $a$  tandis que sa partie imaginaire,  $\Im(z)$ , est  $b$ .

En python, l'unité imaginaire  $i$  est dénotée par la lettre `j`.

```
z = 1+3j
print(z)
```

```
## (1+3j)
```

```
print(type(z))
```

```
## <class 'complex'>
```

Il est également possible d'utiliser la fonction `complex()`, qui demande deux paramètres (la partie réelle et la partie imaginaire) :

```
z = complex(1, 3)
print(z)
```

```
## (1+3j)
```

```
print(type(z))
```

```
## <class 'complex'>
```

Plusieurs méthodes sont disponibles avec les nombres complexes. Par exemple, pour accéder au conjugué, Python fournit la méthode `conjugate()` :

```
print(z.conjugate())
```

```
## (1-3j)
```

L'accès à la partie réelle d'un complexe ou à sa partie imaginaire s'effectue à l'aide des méthodes `real()` et `imag()`, respectivement.

```
z = complex(1, 3)
print(z.real())
```

```
## TypeError: 'float' object is not callable
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print(z.imag())
```

```
## TypeError: 'float' object is not callable
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

## 2.2.4 Conversions

Pour convertir un nombre dans un autre format numérique, Python dispose de quelques fonctions.

### 2.2.4.1 Conversion en entier

La **conversion d'un nombre ou d'une chaîne de caractères en entier** s'effectue à l'aide de la fonction `int()` :

```
x = "3"
x_int = int(x)
print(type(x))
```

```
## <class 'str'>
```

On note que la conversion d'un nombre à virgule flottante tronque le nombre pour ne garder que la partie entière :

```
x = 3.6
x_int = int(x)
print(x_int)
```

```
## 3
```

#### 2.2.4.2 Conversion en nombre à virgule flottante

Pour **convertir un nombre ou une chaîne de caractères en nombre à virgule flottante** (si possible), Python propose d'utiliser la fonction `float()`.

```
x = "3.6"
x_float = float(x)
print(type(x_float))
```

```
## <class 'float'>
```

Avec un entier à l'origine :

```
x = 3
x_float = float(x)
print(x_float)
```

```
## 3.0
```

#### 2.2.4.3 Conversion en complexe

La conversion d'un nombre ou d'une chaîne de caractères en nombre complexe s'effectue avec la fonction `complex()` :

```
x = "2"
x_complex = complex(x)
print(x_complex)
```

```
## (2+0j)
```

Avec un *float* :

```
x = 2.4
x_complex = complex(x)
print(x_complex)
```

```
## (2.4+0j)
```

## 2.3 Booléens

Les données de type logique peuvent prendre deux valeurs : **True** ou **False**. Elles répondent à une condition logique. Il faut faire attention à bien respecter la casse.

```
x = True
y = False
print(x, y)
```

```
## True False
```

**True** peut être converti automatiquement en 1 ; **False** en 0. Cela peut s'avérer très pratique, pour faire des comptages de valeurs vraies ou fausses dans les colonnes d'un tableau de données, par exemple.

```
res = True + True + False + True*True
print(res)
```

```
## 3
```

## 2.4 Objet vide

L'objet vide, communément appelé **null**, possède un équivalent en Python : **None**. Pour l'assigner à une variable, il faut faire attention à la casse :

```
x = None
print(x)
```

```
## None
```

```
print(type(x))
```

```
## <class 'NoneType'>
```

L'objet `None` est une variable neutre, au comportement “null”.

Pour tester si un objet est l'objet `None`, on procède comme suit (le résultat est un booléen) :

```
x = 1
y = None
print(x is None)
```

```
## False
```

```
print(y is None)
```

```
## True
```

## 2.5 Dates et temps

Il existe plusieurs modules pour gérer les dates et le temps en Python. Nous allons explorer une partie du module `datetime`.

### 2.5.1 Module `datetime`

Python possède un module appelé `datetime` qui offre la possibilité de manipuler des dates et des durées (*dates* et *times*).

Il existe plusieurs types d'objets désignant des dates :

- **date** : une date suivant le calendrier grégorien, renseignant l'année, le mois et le jour ;
- **time** : un temp donné, sans prise en compte d'un jour particulier, renseignant l'heure, la minute, la seconde (possiblement la microseconde et le fuseau horaire également).
- **datetime** : une date combinant **date** et **time** ;
- **timedelta** : une durée entre deux objets de type **date**, **time** ou **datetime** ;
- **tzinfo** : un type de base abstraite, renseignant au sujet des fuseaux horaires ;
- **timezone** : un type utilisant le type **tzinfo** comme un décalage fixe par rapport à l'UTC.

#### 2.5.1.1 Date

Les objets de type **date** désignent des dates du calendrier grégorien, pour lesquelles sont mentionnées les caractéristiques suivantes : l'année, le mois et le jour.

Pour créer un objet `date`, la syntaxe est la suivante :

```
date(year, month, day)
```

Par exemple, pour créer la date renseignant le 23 avril 2013 :

```
from datetime import date
debut = date(year = 2013, month = 4, day = 23)
print(debut)
```

```
## 2013-04-23
```

```
print(type(debut))
```

```
## <class 'datetime.date'>
```

#### Remarque 2.5.1

Il n'est pas obligatoire de préciser le nom des paramètres dans l'appel à la fonction `date`. L'ordre à respecter devra toutefois être le suivant : année, mois, jour.

On peut ensuite accéder aux attributs de la date créée (ce sont des entiers) :

```
print(debut.year) # Extraire l'année
```

```
## 2013
```

```
print(debut.month) # Extraire le mois
```

```
## 4
```

```
print(debut.day) # Extraire le jour
```

```
## 23
```

Les objets du type `date` possèdent quelques méthodes. Nous allons passer en revue quelques-unes d'entre-elles.

##### 2.5.1.1.1 `ctime()`

La méthode `ctime()` retourne la date sous forme d'une chaîne de caractères.

```
debut.ctime()
```

#### 2.5.1.1.2 `weekday()`

La méthode `weekday()` retourne la position du jour de la semaine (lundi valant 0, dimanche 6)

```
debut.weekday()
```

#### Remarque 2.5.2

Cette méthode peut être très pratique lors d'une analyse des données, pour explorer les aspects de saisonnalité hebdomadaire.

#### 2.5.1.1.3 `isoweekday()`

Dans la même veine que `weekday()`, la méthode `isoweekday()` retourne la position du jour de la semaine, en attribuant cette fois la valeur 1 au lundi et 7 au dimanche.

```
debut.isoweekday()
```

#### 2.5.1.1.4 `toordinal()`

La méthode `toordinal()` retourne le numéro du jour, en prenant comme référence la valeur 1 pour le premier jour de l'an 1.

```
debut.toordinal()
```

#### 2.5.1.1.5 `isoformat()`

La méthode `isoformat()` retourne la date en [numérotation ISO](#), sous forme d'une chaîne de caractères.

```
debut.isoformat()
```

#### 2.5.1.1.6 `isocalendar()`

La méthode `isocalendar()` retourne un nuplet (c.f. Section [3.2](#)) comprenant trois éléments : l'année, le numéro de la semaine et le jour de la semaine (les trois en numérotation ISO).

```
debut.isocalendar()
```

#### 2.5.1.1.7 `replace()`

La méthode `replace()` retourne la date après avoir effectué une modification

```
x = debut.replace(year=2014)
y = debut.replace(month=5)
z = debut.replace(day=24)
print(x, y, z)

## 2014-04-23 2013-05-23 2013-04-24
```

Cela n'a pas d'incidence sur l'objet d'origine :

```
print(debut)

## 2013-04-23
```

Il est possible de modifier plusieurs éléments en même temps :

```
x = debut.replace(day=24, month=5)
print(x)

## 2013-05-24
```

#### 2.5.1.1.8 `strftime()`

La méthode `strftime()` retourne, sous la forme d'une chaîne de caractères, une représentation de la date, selon un masque utilisé.

Par exemple, pour que la date soit représentée sous la forme DD-MM-YYYY (jour sur deux chiffres, mois sur deux chiffres et année sur 4) :

```
print(debut.strftime("%d-%m-%Y"))

## 23-04-2013
```

Dans l'exemple précédent, on note deux choses : la présence de directives de formatage (qui commencent par le symbole de pourcentage) et des caractères autres (ici, les tirets). On peut noter que les caractères peuvent être remplacés par d'autres, il s'agit ici d'un choix pour représenter la date en séparant ses éléments par des tirets. Il est tout à fait possible d'adopter une autre écriture, par exemple avec des barres obliques, ou même d'autres chaînes de caractères :



```
print(debut.strftime("%d/%m/%Y"))
```

```
## 23/04/2013
```

```
print(debut.strftime("Jour : %d, Mois : %m, Annee : %Y"))
```

```
## Jour : 23, Mois : 04, Annee : 2013
```

Concernant les directives de formatage, elles correspondent aux codes requis par le standard C (c.f. la [documentation de Python](#)). En voici quelques-uns :

TABLE 2.3 – Codes de formatages

Code	Description	Exemple
%a	Abréviation du jour de la semaine (dépend du lieu)	Tue
%A	Jour de la semaine complet (dépend du lieu)	Tuesday
%b	Abréviation du mois (dépend du lieu)	Apr
%B	Nom du mois complet (dépend du lieu) octobre	April
%c	Date et heure (dépend du lieu) au format %a %e %b %H : %M : %S : %Y	Tue Apr 23 00:00:00 2013
%C	Siècle (00-99) -1 (partie entière de la division de l'année par 100)	20
%d	Jour du mois (01-31)	23
%D	Date au format %m/%d/%y	04/23/13
%e	Jour du mois en nombre décimal (1-31)	23
%F	Date au format %Y-%m-%d	2013-04-23
%h	Même chose que %b	Apr
%H	Heure (00-24)	00
%I	Heure (01-12)	12
%j	Jour de l'année (001-366)	113
%m	Mois (01-12)	04
%M	Minute (00-59)	00
%n	Retour à la ligne en output, caractère blanc en input	\n
%p	AM/PM PM	AM
%r	Heure au format 12 AM/PM	12:00:00 AM
%R	Même chose que %H : %M	00:00
%S	Seconde (00-61)	00
%t	Tabulation en output, caractère blanc en input	\t
%T	Même chose que %H : %M : %S	00:00:00
%u	Jour de la semaine (1-7), commence le lundi	2
%U	Semaine de l'année (00-53), dimanche comme début de semaine, et le premier dimanche de l'année définit la semaine	16

Code	Description	Exemple
%V	Semaine de l'année (00-53). Si la semaine (qui commence un lundi) qui contient le 1 <sup>er</sup> janvier a quatre jours ou plus dans la nouvelle année, alors elle est considérée comme la semaine 1. Sinon, elle est considérée comme la dernière de l'année précédente, et la semaine suivante est considérée comme semaine 1 (norme ISO 8601)	17
%w	Jour de la semaine (0-6), dimanche étant 0	2
%W	Semaine de l'année (00-53), le lundi étant le premier jour de la semaine, et typiquement, le premier lundi de l'année définit la semaine 1 (convention G.B.)	16
%x	Date (dépend du lieu)	04/23/13
%X	Heure (dépend du lieu)	00:00:00'
%y	Année sans le "siècle" (00-99)	13
%Y	Année (en input, uniquement de 0 à 9999)	2013
%z	offset en heures et minutes par rapport au temps UTC	
%Z	Abréviation du fuseau horaire (en output seulement) CEST	

### 2.5.1.2 Time

Les objets de type `time` désignent des temps précis sans prise en compte d'un jour particulier. Ils renseignent l'heure, la minute, la seconde (possiblement la microseconde et le fuseau horaire également).

Pour créer un objet `time`, la syntaxe est la suivante :

```
time(hour, minute, second)
```

Par exemple, pour créer le moment 23 :04 :59 (vingt-trois heures, quatre minutes et cinquante-neuf secondes) :

```
from datetime import time
moment = time(hour = 23, minute = 4, second = 59)
print(moment)
```

```
## 23:04:59
```

```
print(type(moment))
```

```
## <class 'datetime.time'>
```

On peut rajouter des informations sur la microseconde. Sa valeur doit être comprise entre zéro et un million.

```
moment = time(hour = 23, minute = 4, second = 59, microsecond = 230)
print(moment)
```

```
## 23:04:59.000230
```

```
print(type(moment))
```

```
## <class 'datetime.time'>
```

On peut ensuite accéder aux attributs de la date créée (ce sont des entiers), parmi lesquels :

```
print(moment.hour) # Extraire l'heure
```

```
## 23
```

```
print(moment.minute) # Extraire la minute
```

```
## 4
```

```
print(moment.second) # Extraire la seconde
```

```
## 59
```

```
print(moment.microsecond) # Extraire la microseconde
```

```
## 230
```

Les objets du type `time` possèdent quelques méthodes, dont l'utilisation est similaire aux objets de classe `date` (se référer à la Section [2.5.1.1](#)).

### 2.5.1.3 Datetime

Les objets de type `datetime` combinent les éléments des objets de type `date` et `time`. Ils renseignent le jour dans le calendrier grégorien ainsi que l'heure, la minute, la seconde (possiblement la microseconde et le fuseau horaire).

Pour créer un objet `datetime`, la syntaxe est la suivante :

```
datetime(year, month, day, hour, minute, second, microsecond)
```

Par exemple, pour créer la date 23-04-2013 à 17 :10 :00 :

```
from datetime import datetime
x = datetime(year = 2013, month = 4, day = 23,
             hour = 23, minute = 4, second = 59)
print(x)
```

```
## 2013-04-23 23:04:59
```

```
print(type(x))
```

```
## <class 'datetime.datetime'>
```

Les objets de type `datetime` disposent des attributs des objets de type `date` (c.f. Section 2.5.1.1) et de type `time` (c.f. Section 2.5.1.2).

Pour ce qui est des méthodes, davantage sont disponibles. Nous allons en commenter certaines.

#### 2.5.1.3.1 `today()` et `now()`

Les méthodes `today()` et `now()` retournent le `datetime` courant, celui au moment où est évaluée l'instruction :

```
print(x.today())
```

```
## 2018-10-08 22:38:37.319853
```

```
print(datetime.today())
```

```
## 2018-10-08 22:38:37.324472
```

La distinction entre les deux réside dans le fuseau horaire. Avec `today()`, l'attribut `tzinfo` est mis à `None`, tandis qu'avec `now()`, l'attribut `tzinfo`, s'il est indiqué, est pris en compte.

#### 2.5.1.3.2 `timestamp()`

La méthode `timestamp()` retourne, sous forme d'un nombre à virgule flottante, le *timestamp* POSIX correspondant à l'objet de type `datetime`. Le *timestamp* POSIX correspond à l'heure Posix, équivalent au nombre de secondes écoulées depuis le premier janvier 1970, à 00 :00 :00 UTC.

```
print(x.timestamp())
```

```
## 1366751099.0
```

#### 2.5.1.3.3 date()

La méthode `date()` retourne un objet de type `date` dont les attributs d'année, de mois et de jour sont identiques à ceux de l'objet :

```
x_date = x.date()  
print(x_date)
```

```
## 2013-04-23
```

```
print(type(x_date))
```

```
## <class 'datetime.date'>
```

#### 2.5.1.3.4 time()

La méthode `time()` retourne un objet de type `time` dont les attributs d'heure, minute, seconde, microseconde sont identiques à ceux de l'objet :

```
x_time = x.time()  
print(x_time)
```

```
## 23:04:59
```

```
print(type(x_time))
```

```
## <class 'datetime.time'>
```

#### 2.5.1.4 Timedelta

Les objets de type `timedelta` représentent des durées séparant deux dates ou heures.

Pour créer un objet de type `timedelta`, la syntaxe est la suivante :

```
timedelta(days, hours, minutes, seconds, microseconds)
```

Il n'est pas obligatoire de fournir une valeur à chaque paramètre. Lorsque qu'un paramètre ne reçoit pas de valeur, celle qui lui est attribuée par défaut est 0.

Par exemple, pour créer un objet indiquant une durée de 1 jour et 30 secondes :

```
from datetime import timedelta
duree = timedelta(days = 1, seconds = 30)
duree
```

```
datetime.timedelta(1, 30)
```

On peut accéder ensuite aux attributs (ayant été définis). Par exemple, pour accéder au nombre de jours que représente la durée :

```
duree.days
```

```
1
```

La méthode `total_seconds()` permet d'obtenir la durée exprimée en secondes :

```
duree = timedelta(days = 1, seconds = 30, hours = 20)
duree.total_seconds()
158430.0
```

#### 2.5.1.4.1 Durée séparant deux objets `date` ou `datetime`

Lorsqu'on soustrait deux objets de type `date`, on obtient le nombre de jours séparant ces deux dates, sous la forme d'un objet de type `timedelta` :

```
from datetime import timedelta
debut = date(2018, 1, 1)
fin = date(2018, 1, 2)
nb_jours = fin - debut
print(type(nb_jours))

## <class 'datetime.timedelta'>
```

```
print(nb_jours)
```

```
## 1 day, 0:00:00
```

Lorsqu'on soustrait deux objets de type `datetime`, on obtient le nombre de jours, secondes (et microsecondes, si renseignées) séparant ces deux dates, sous la forme d'un objet de type `timedelta` :

```
debut = datetime(2018, 1, 1, 12, 26, 30, 230)
fin = datetime(2018, 1, 2, 11, 14, 31)
duree = fin - debut
print(type(duree))
```

```
## <class 'datetime.timedelta'>
```

```
print(duree)
```

```
## 22:48:00.999770
```

On peut noter que les durée données prennent en compte les années bissextiles. Regardons d'abord pour une année non-bissextile, le nombre de jours séparant le 28 février du premier mars :

```
debut = date(2021, 2, 28)
fin = date(2021, 3, 1)
duree = fin - debut
duree
```

```
datetime.timedelta(1)
```

Regardons à présent la même chose, mais dans le cas d'une année bissextile :

```
debut_biss = date(2020, 2, 28)
fin_biss = date(2020, 3, 1)
duree_biss = fin_biss - debut_biss
duree_biss
```

```
datetime.timedelta(2)
```

Il est également possible d'**ajouter des durées à une date** :

```
debut = datetime(2018, 12, 31, 23, 59, 59)
print(debut + timedelta(seconds = 1))
```

```
## 2019-01-01 00:00:00
```

## 2.5.2 Module pytz

Si la gestion des dates revêt une importance particulière, une librairie propose d'aller un peu plus loin, notamment en ce qui concerne la gestion des fuseaux horaires. Cette librairie s'appelle **pytz**. De nombreux exemples sont proposés sur [la page web du projet](#).

### 2.5.3 Exercices

1. En utilisant la fonction appropriée, stocker la date du 29 août 2019 dans un objet que l'on appellera `d` puis afficher le type de l'objet.
2. À l'aide de la fonction appropriée, afficher la date du jour.
3. Stocker la date suivante dans un objet nommé `d2` : “2019-08-29 20 :30 :56”. Puis, afficher dans la console avec la fonction `print()` les attributs d'année, de minute et de seconde de `d2`.
4. Ajouter 2 jours, 3 heures et 4 minutes à `d2`, et stocker le résultat dans un objet appelé `d3`.
5. Afficher la différence en secondes entre `d3` et `d2`.
6. À partir de l'objet `d2`, afficher sous forme de chaîne de caractères la date de `d2` de manière à ce qu'elle respecte la syntaxe suivante : “Mois Jour, Année”, avec “Mois” le nom du mois (August), “Jour” le numéro du jour sur deux chiffres (29) et “Année” l'année de la date (2019).



# Chapitre 3

## Structures

Python dispose de plusieurs structures différentes intégrées de base. Nous allons aborder dans cette partie quelques unes d'entre-elles : les listes, les N-uplet (ou *tuples*), les ensembles et les dictionnaires.

### 3.1 Listes

Une des structures les plus flexibles en Python est la liste. Il s'agit d'un regroupement de valeurs. La création d'une liste s'effectue en écrivant les valeurs en les séparant par une virgule et en entourant l'ensemble par des crochets ([ et ]).

```
x = ["Pascaline", "Gauthier", "Xuan", "Jimmy"]
print(x)

## ['Pascaline', 'Gauthier', 'Xuan', 'Jimmy']
```

Le contenu d'une liste n'est pas forcément du texte :

```
y = [1, 2, 3, 4, 5]
print(y)

## [1, 2, 3, 4, 5]
```

Il est même possible de faire figurer des éléments de type différent dans une liste :

```
z = ["Piketty", "Thomas", 1971]
print(z)

## ['Piketty', 'Thomas', 1971]
```

Une liste peut contenir une autre liste :

```
tweets = ["aaa", "bbb"]
followers = ["Anne", "Bob", "Irma", "John"]
compte = [tweets, followers]
print(compte)

## [['aaa', 'bbb'], ['Anne', 'Bob', 'Irma', 'John']]
```

### 3.1.1 Extraction des éléments

L'accès aux éléments se fait grace à son indexation (attention, l'indice du premier élément est 0) :

```
print(x[0]) # Le premier élément de x
```

```
## Pascaline
```

```
print(x[1]) # Le second élément de x
```

```
## Gauthier
```

L'accès à un élément peut aussi se faire en parant de la fin, en faisant figurer le signe moins (-) devant l'indice : L'accès aux éléments se fait grace à son indexation (attention, l'indice du premier élément est 0) :

```
print(x[-1]) # Le dernier élément de x
```

```
## Jimmy
```

```
print(x[-2]) # L'avant dernier élément de x
```

```
## Xuan
```

Le découpage d'une liste de manière à obtenir un sous-ensemble de la liste s'effectue avec les deux points (:) :

```
print(x[1:2]) # Les premiers et seconds éléments de x
```

```
## ['Gauthier']
```

```
print(x[2:]) # Du second (non inclus) à la fin de x
```

```
## ['Xuan', 'Jimmy']
```

```
print(x[:-2]) # Du premier à l'avant dernier (non inclus)
```

```
## ['Pascaline', 'Gauthier']
```

#### Remarque 3.1.1

Le découpage retourne également une liste.

Lors de l'extraction des éléments de la liste à l'aide des crochets, il est possible de rajouter un troisième paramètre, le pas :

```
print(x[::2]) # Un élément sur deux
```

```
## ['Pascaline', 'Xuan']
```

L'accès à des listes imbriquées s'effectue en utilisant plusieurs fois les crochets :

```
tweets = ["aaa", "bbb"]  
followers = ["Anne", "Bob", "Irma", "John"]  
compte = [tweets, followers]  
res = compte[1][3] # Le 4e élément du 2e élément de la liste compte
```

Le nombre d'éléments d'une liste s'obtient avec la fonction `len()` :

```
print(len(compte))
```

```
## 2
```

```
print(len(compte[1]))
```

```
## 4
```

### 3.1.2 Modification

Les listes sont mutables, c'est-à-dire que leur contenu peut être modifié une fois l'objet créé.

### 3.1.2.1 Remplacement

Pour **modifier** un élément dans une liste, on utilise l'indiciage :

```
x = [1, 3, 5, 6, 9]
x[3] = 7 # Remplacement du 4e élément
print(x)

## [1, 3, 5, 7, 9]
```

### 3.1.2.2 Ajout d'éléments

Pour **ajouter des éléments à une liste**, on utilise la méthode `append()` :

```
x.append(11) # Ajout de la valeur 11 en fin de liste
print(x)

## [1, 3, 5, 7, 9, 11]
```

Il est aussi possible d'utiliser la méthode `extend()`, pour concaténer des listes :

```
y = [13, 15]
x.extend(y)
print(x)

## [1, 3, 5, 7, 9, 11, 13, 15]
```

### 3.1.2.3 Suppression d'éléments

Pour **retirer un élément d'une liste**, on utilise la méthode `remove()` :

```
x.remove(3) # Retire le 4e élément
print(x)

## [1, 5, 7, 9, 11, 13, 15]
```

On peut aussi utiliser la commande `del` :

```
x = [1, 3, 5, 6, 9]
del x[3] # Retire le 4e élément
print(x)
```

```
## [1, 3, 5, 9]
```

### 3.1.2.4 Affectations multiples

On peut modifier plusieurs valeurs en même temps :

```
x = [1, 3, 5, 6, 10]
x[3:5] = [7, 9] # Remplace les 4e et 5e valeurs
print(x)
```

```
## [1, 3, 5, 7, 9]
```

La modification peut agrandir la taille de la liste :

```
x = [1, 2, 3, 4, 5]
x[2:3] = ['a', 'b', 'c', 'd'] # Remplace la 3e valeur
print(x)
```

```
## [1, 2, 'a', 'b', 'c', 'd', 4, 5]
```

On peut supprimer plusieurs valeurs en même temps :

```
x = [1, 2, 3, 4, 5]
x[3:5] = [] # Retire les 4e et 5e valeurs
print(x)
```

```
## [1, 2, 3]
```

### 3.1.3 Test d'appartenance

En utilisant l'opérateur `in`, on peut tester l'appartenance d'un objet à une liste :

```
x = [1, 2, 3, 4, 5]
print(1 in x)
```

```
## True
```

### 3.1.4 Copie de liste

Attention, la copie d'une liste n'est pas triviale en Python. Prenons un exemple.

```
x = [1, 2, 3]
y = x
```

Modifions le premier élément de y, et observons le contenu de y et de x :

```
y[0] = 0
print(y)

## [0, 2, 3]
```

```
print(x)

## [0, 2, 3]
```

Comme on peut le constater, le fait d'avoir utilisé le signe égal a simplement créé une référence et non pas une copie.

Pour effectuer une copie de liste, plusieurs façons existent. Parmi elles, l'utilisation de la fonction `list()` :

```
x = [1, 2, 3]
y = list(x)
y[0] = 0
print("x : ", x)

## x : [1, 2, 3]
```

```
print("y : ", y)

## y : [0, 2, 3]
```

On peut noter que lorsque l'on fait un découpage, un nouvel objet est créé, pas une référence :

```
x = [1, 2, 3, 4]
y = x[:2]
y[0] = 0
print("x : ", x)

## x : [1, 2, 3, 4]
```

```
print("y : ", y)

## y : [0, 2]
```

### 3.1.5 Tri

Pour trier les objets de la liste (sans en créer une nouvelle), Python propose la méthode `sort()` :

```
x = [2, 1, 4, 3]
x.sort()
print(x)

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

Cela fonctionne également avec des valeurs textuelles, en triant par ordre alphabétique :

```
x = ["c", "b", "a", "a"]
x.sort()
print(x)

## ['a', 'a', 'b', 'c']
```

Il est possible de fournir à la méthode `sort()` des paramètres. Parmi ces paramètres, il en est un, `key`, qui permet de fournir une fonction pour effectuer le tri. Cette fonction doit retourner une valeur pour chaque objet de la liste, sur laquelle le tri sera effectué. Par exemple, avec la fonction `len()`, qui, lorsqu'appliquée à du texte, retourne le nombre de caractères :

```
x = ["aa", "a", "aaaaa", "aa"]
x.sort(key=len)
print(x)

## ['a', 'aa', 'aa', 'aaaaa']
```

## 3.2 N-uplets (Tuples)

Les n-uplets, ou *tuples* sont des séquences d'objets Python.

Pour créer un n-uplet, on liste les valeurs, séparées par des virgules :

```
x = 1, 4, 9, 16, 25
print(x)

## (1, 4, 9, 16, 25)
```

On note que les n-uplets sont repérés par une suite de valeurs, entourées dans deux parenthèses.

### 3.2.1 Extraction des éléments

Les éléments d'un n-uplet s'extraient de la même manière que ceux des listes (c.f. Section 3.1.1).

```
print(x[0])

## 1
```

### 3.2.2 Modification

Contrairement aux listes, les n-uplets sont **inaltérables** (c'est-à-dire ne pouvant pas être modifiés après avoir été créés) :

```
x[0] = 1

## TypeError: 'tuple' object does not support item assignment
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Il est possible d'**imbriquer des n-uplets** à l'intérieur d'un autre n-uplet. Pour ce faire, on a recours à l'utilisation de parenthèses :

```
x = ((1, 4, 9, 16), (1, 8, 26, 64))
print(x)

## ((1, 4, 9, 16), (1, 8, 26, 64))
```

## 3.3 Ensembles

Les ensembles (*sets*) sont des collections non ordonnée d'éléments uniques. Les ensembles sont inaltérables, et non indexés.



Pour créer un ensemble, Python fournit la fonction `set()`. On fournit un ou plusieurs éléments constituant l'ensemble, en les séparant par des virgules et en entourant l'ensemble d'accolades (`{}`) :

```
ensemble = set({"Marseille", "Aix-en-Provence", "Nice", "Rennes"})
print(ensemble)
```

```
## {'Nice', 'Rennes', 'Marseille', 'Aix-en-Provence'}
```

De manière équivalente, on peut ne pas utiliser la fonction `set()` et définir l'ensemble uniquement à l'aide des crochets :

```
ensemble = {"Marseille", "Aix-en-Provence", "Nice", "Rennes"}
print(ensemble)
```

```
## {'Nice', 'Rennes', 'Marseille', 'Aix-en-Provence'}
```

En revanche, si l'ensemble est vide, Python retourne un erreur si la fonction `set()` n'est pas utilisée : il est nécessaire d'utiliser la fonction `set` :

```
ensemble_vide = {}
type(ensemble_vide)
```

Le type de l'objet que l'on vient de créer n'est pas `set` mais `dict` (c.f. Section 3.4). Aussi, pour créer l'ensemble vide, on utilise `set()` :

```
ensemble_vide = set()
type(ensemble_vide)
```

Lors de la création, s'il existe des doublons dans les valeurs fournies, ils seront supprimés pour ne garder qu'une seule valeur :

```
ensemble = set({"Marseille", "Aix-en-Provence", "Nice", "Marseille", "Rennes"})
print(ensemble)
```

```
## {'Nice', 'Rennes', 'Marseille', 'Aix-en-Provence'}
```

La longueur d'un ensemble s'obtient à l'aide de la fonction `len()` :

```
print(len(ensemble))
```

```
## 4
```

### 3.3.1 Modifications

#### 3.3.1.1 Ajout

Pour ajouter un élément à un ensemble, Python offre la méthode `add()` :

```
ensemble.add("Toulon")
print(ensemble)
```

```
## {'Rennes', 'Nice', 'Toulon', 'Aix-en-Provence', 'Marseille'}
```

Si l'élément est déjà présent, il ne sera pas ajouté :

```
ensemble.add("Toulon")
print(ensemble)
```

```
## {'Rennes', 'Nice', 'Toulon', 'Aix-en-Provence', 'Marseille'}
```

#### 3.3.1.2 Suppression

Pour supprimer une valeur d'un ensemble, Python propose la méthode `remove()` :

```
ensemble.remove("Toulon")
print(ensemble)
```

```
## {'Rennes', 'Nice', 'Aix-en-Provence', 'Marseille'}
```

Si la valeur n'est pas présente dans l'ensemble, Python retourne un message d'erreur :

```
ensemble.remove("Toulon")
```

```
## KeyError: 'Toulon'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print(ensemble)
```

```
## {'Rennes', 'Nice', 'Aix-en-Provence', 'Marseille'}
```

### 3.3.2 Test d'appartenance

Un des intérêts des ensembles est la recherche rapide de présence ou absence de valeurs (plus rapide que dans une liste). Comme pour les listes, les tests d'appartenance s'effectuent à l'aide de l'opérateur `in` :

```
print("Marseille" in ensemble)
```

```
## True
```

```
print("Paris" in ensemble)
```

```
## False
```

### 3.3.3 Copie d'ensemble

Pour copier un ensemble, comme pour les listes (c.f. Section 3.1.4), il ne faut pas utiliser le signe d'égalité. La copie d'un ensemble se fait à l'aide de la méthode `copy()` :

```
ensemble = set({"Marseille", "Aix-en-Provence", "Nice"})
y = ensemble.copy()
y.add("Toulon")
print("y : ", y)
```

```
## y :  {'Nice', 'Toulon', 'Marseille', 'Aix-en-Provence'}
```

```
print("ensemble : ", ensemble)
```

```
## ensemble :  {'Nice', 'Marseille', 'Aix-en-Provence'}
```

### 3.3.4 Conversion en liste

Un des intérêts des ensembles est qu'ils contiennent des éléments uniques. Aussi, lorsque l'on souhaite obtenir les éléments distincts d'une liste, il est possible de la convertir en ensemble (avec la fonction `set()`), puis de convertir l'ensemble en liste (avec la fonction `list()`) :

```
ma_liste = ["Marseille", "Aix-en-Provence", "Marseille", "Marseille"]
print(ma_liste)
```

```
## ['Marseille', 'Aix-en-Provence', 'Marseille', 'Marseille']
```

```
mon_ensemble = set(ma_liste)
print(mon_ensemble)
```

```
## {'Marseille', 'Aix-en-Provence'}
```

```
ma_nouvelle_liste = list(mon_ensemble)
print(ma_nouvelle_liste)
```

```
## ['Marseille', 'Aix-en-Provence']
```

## 3.4 Dictionnaires

Les dictionnaires en Python sont une implémentation d'objets clé-valeurs, les clés étant indexées.

Les clés sont souvent du texte, les valeurs peuvent être de différents types et différentes structures.

Pour créer un dictionnaire, on peut procéder en utilisant des accolades (`{}`). Comme rencontré dans la Section 3.3, si on évalue le code suivant, on obtient un dictionnaire :

```
dict_vide = {}
print(type(dict_vide))
```

```
## <class 'dict'>
```

Pour créer un dictionnaire avec des entrées, on peut utiliser les accolades, on sépare chaque entrée par des virgules, et on distingue la clé de la valeur associée par deux points (`:`) :

```
mon_dict = { "nom": "Kyrie",
             "prenom": "John",
             "naissance": 1992,
             "equipes": ["Cleveland", "Boston"]}
print(mon_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Cleveland', 'Boston']}
```

Il est aussi possible de créer un dictionnaire à l'aide de la fonction `dict()`, en fournissant une séquence de clés-valeurs :

```
x = dict([("Julien-Yacine", "Data-scientist"),
          ("Sonia", "Directrice")])
print(x)

## {'Julien-Yacine': 'Data-scientist', 'Sonia': 'Directrice'}
```

### 3.4.1 Extraction des éléments

L'extraction dans les dictionnaires repose sur le même principe que pour les listes et les n-uplets (c.f. Section @ref(#structure-liste-extraction)). Toutefois, l'extraction d'un élément d'un dictionnaire ne se fait pas en fonction de sa position dans le dictionnaire, mais par sa clé :

```
print(mon_dict["prenom"])
```

```
## John
```

```
print(mon_dict["equipes"])
```

```
## ['Cleveland', 'Boston']
```

Si l'extraction s'effectue par une clé non présente dans le dictionnaire, une erreur sera retournée :

```
print(mon_dict["age"])
```

```
## KeyError: 'age'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

On peut tester la présence d'une clé avec l'opérateur `in` :

```
print("prenom" in mon_dict)
```

```
## True
```

```
print("age" in mon_dict)
```

```
## False
```

L'extraction de valeurs peut aussi se faire à l'aide de la méthode `get()`, qui retourne une valeur `None` si la clé n'est pas présente :

```
print(mon_dict.get("prenom"))
```

```
## John
```

```
print(mon_dict.get("age"))
```

```
## None
```

### 3.4.2 Clés et valeurs

À l'aide de la méthode `key()`, on peut accéder aux clés du dictionnaire :

```
les_cles = mon_dict.keys()
print(les_cles)
```

```
## dict_keys(['nom', 'prenom', 'naissance', 'equipes'])
```

```
print(type(les_cles))
```

```
## <class 'dict_keys'>
```

Il est possible par la suite de transformer cette énumération de clés en liste :

```
les_cles_liste = list(les_cles)
print(les_cles_liste)
```

```
## ['nom', 'prenom', 'naissance', 'equipes']
```

La méthode `values()` fournit quand à elle les valeurs du dictionnaire :

```
les_valeurs = mon_dict.values()
print(les_valeurs)
```

```
## dict_values(['Kyrie', 'John', 1992, ['Cleveland', 'Boston']])
```

```
print(type(les_valeurs))
```

```
## <class 'dict_values'>
```

La méthode `items()` fournit quand à elle les clés et valeurs sous forme de n-uplets :

```
les_items = mon_dict.items()
print(les_items)
```

```
## dict_items([('nom', 'Kyrie'), ('prenom', 'John'), ('naissance',
    1992), ('equipes', ['Cleveland', 'Boston'])])
```

```
print(type(les_items))
```

```
## <class 'dict_items'>
```

### 3.4.3 Recherche d'appartenance

Grâce aux méthodes `keys()`, `values()` et `items()`, il est aisé de rechercher la présence d'objets dans un dictionnaire.

```
print("age" in les_cles)
```

```
## False
```

```
print("nom" in les_cles)
```

```
## True
```

```
print(['Cleveland', 'Boston'] in les_valeurs)
```

```
## True
```

### 3.4.4 Modification

#### 3.4.4.1 Remplacement

Pour remplacer la valeur associée à une clé, on peut utiliser les crochets ([]) et le signe d'égalité (=).

Par exemple, pour remplacer les valeurs associées à la clé `equipes` :

```
mon_dict["equipes"] = ["Montclair Kimberley Academy",
    "Cleveland Cavaliers", "Boston Celtics"]
print(mon_dict)

## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics']}
```

#### 3.4.4.2 Ajout d'éléments

L'ajout d'un élément dans un dictionnaire peut s'effectuer avec les crochets ([]) et le signe d'égalité (=) :

```
mon_dict["taille_cm"] = 191
print(mon_dict)

## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics'], 'taille_cm': 191}
```

Pour ajouter le contenu d'un autre dictionnaire à un dictionnaire, Python propose la méthode `update()`.

Créons un second dictionnaire dans un premier temps :

```
second_dict = {"masse_kg" : 88, "debut_nba" : 2011}
print(second_dict)

## {'masse_kg': 88, 'debut_nba': 2011}
```

Ajoutons le contenu de ce second dictionnaire au premier :

```
mon_dict.update(second_dict)
print(mon_dict)
```



```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics'], 'taille_cm': 191, 'masse_kg': 88, 'debut_nba': 2011}
```

Si on modifie par la suite le second dictionnaire, cela n'aura pas d'incidence sur le premier :

```
second_dict["poste"] = "PG"
print(second_dict)
```

```
## {'masse_kg': 88, 'debut_nba': 2011, 'poste': 'PG'}
```

```
print(mon_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics'], 'taille_cm': 191, 'masse_kg': 88, 'debut_nba': 2011}
```

### 3.4.4.3 Suppression d'éléments

La suppression d'un élément dans un dictionnaire peut s'effectuer de plusieurs manières. Par exemple, avec l'opérateur `del` :

```
del mon_dict["debut_nba"]
print(mon_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics'], 'taille_cm': 191, 'masse_kg': 88}
```

Il est également possible d'utiliser la méthode `pop()` :

```
res = mon_dict.pop("masse_kg")
print(mon_dict)
```

```
## {'nom': 'Kyrie', 'prenom': 'John', 'naissance': 1992, 'equipes':
    ['Montclair Kimberley Academy', 'Cleveland Cavaliers', 'Boston
    Celtics'], 'taille_cm': 191}
```

Dans l'instruction précédente, nous avons ajouté une assignation du résultat de l'application de la méthode `pop()` à une variable nommée `res`. Comme on peut le constater, la méthode `pop()`, en plus d'avoir supprimé la clé, a retourné la valeur associée :

```
print(res)
```

```
## 88
```

### 3.4.5 Copie de dictionnaire

Pour copier un dictionnaire, et non créer une référence (ce qui est le cas si on utilise le signe d'égalité), Python fournit comme pour les ensembles, une méthode `copy()` :

```
d = {"Marseille": 13, "Rennes" : 35}
d2 = d.copy()
d2["Paris"] = 75
print("d: ", d)
```

```
## d:  {'Marseille': 13, 'Rennes': 35}
```

```
print("d2: ", d2)
```

```
## d2:  {'Marseille': 13, 'Rennes': 35, 'Paris': 75}
```

### 3.4.6 Exercice

1. Créer un dictionnaire nommé `photo`, comprenant les couples clés-valeurs suivants :
2. clé : `id`, valeur : 1,
3. clé : `description`, valeur : Une photo du Vieux-port de Marseille,
4. clé : `loc`, valeur : une liste dans laquelle sont données les coordonnées suivantes 5.3772133, 43.302424. 2. Ajouter le couple de clé-valeur suivant au dictionnaire `photo` :  
clé : `utilisateur`, valeur : `bob`.
5. Rechercher s'il existe une entrée dont la clé vaut `description` dans le dictionnaire `photo`. Si tel est le cas, afficher l'entrée correspondante (clé et valeur).
6. Supprimer l'entrée dans `photo` dont la clé vaut `utilisateur`.
7. Modifier la valeur de l'entrée `loc` dans le dictionnaire `photo`, pour proposer une nouvelle liste, dont les coordonnées sont les suivantes : 5.3692712 et 43.2949627.

# Chapitre 4

## Opérateurs

Python comprend différents opérateurs, permettant d'effectuer des opérations entre les opérandes, c'est-à-dire entre des variables, des littéraux ou encore des expressions.

### 4.1 Opérateurs arithmétiques

Les opérateurs arithmétiques de base sont intégrés dans Python.

Nous avons déjà utilisé dans les chapitres précédents certains d'entre eux, pour effectuer des opérations sur les entiers ou les nombres à virgule flottante (addition, soustraction, etc.). Faisons un tour rapide des opérateurs arithmétiques les plus courants permettant de réaliser des opérations sur des nombres.

#### 4.1.1 Addition

On effectue une addition entre deux nombres à l'aide du symbole + :

```
print(1+1) # Addition
```

```
## 2
```

#### 4.1.2 Soustraction

On effectue une soustraction entre deux nombres à l'aide du symbole - :

```
print(1-1) # Soustraction
```

```
## 2
```

### 4.1.3 Multiplication

On effectue une multiplication entre deux nombres à l'aide du symbole `*` :

```
print(2*2) # Multiplication
```

```
## 4
```

### 4.1.4 Division

On effectue une division (réelle) entre deux nombres à l'aide du symbole `/` :

```
print(3/2) # Division
```

```
## 1.5
```

Pour effectuer une division entière, on double la barre oblique :

```
print(3//2) # Division entière
```

```
## 1
```

### 4.1.5 Modulo

Le modulo (reste de la division euclidienne) s'obtient à l'aide du symbole `%` :

```
print(12%10) # Modulo
```

```
## 2
```

### 4.1.6 Puissance

Pour élever un nombre à une puissance donnée, on utilise deux étoiles (`**`) :

```
print(2**3) # 2 élevé à la puissance 3
```

```
## 8
```

### 4.1.7 Ordre

L'ordre des opérations suit la règle PEMDAS (*Parentheses, Exponents, Multiplication and Division, Addition and Subtraction*).

Par exemple, l'instruction suivante effectue d'abord le calcul  $2 \times 2$ , puis ajoute 1 :

```
print(2*2+1)
```

```
## 5
```

L'instruction suivante, grâce aux parenthèses, effectue d'abord le calcul  $2 + 1$ , puis la multiplication du résultat avec 2 :

```
print(2*(2+1))
```

```
## 6
```

### 4.1.8 Opérateurs mathématiques sur des chaînes de caractères

Certains opérateurs mathématiques présentés dans la Section 4.1 peuvent-être appliqués à des chaînes de caractères.

Lorsque l'on utilise le symbole  $+$  entre deux chaînes de caractères, Python concatène ces deux chaînes (cf. Section 2.1.1) :

```
a = "euro"
b = "dollar"
print(a+b)
```

```
## eurodollar
```

Lorsqu'on “multiplie” une chaîne par un scalaire  $n$ , Python répète la chaîne le nombre  $n$  fois :

```
2*a
```

### 4.1.9 Opérateurs mathématiques sur des listes ou des n-uplets

Certains opérateurs mathématiques peuvent également être appliquées à des listes.

Lorsque l'on utilise le symbole `+` entre deux listes, Python les concatène en une seule :

```
l_1 = [1, "pomme", 5, 7]
l_2 = [9, 11]
print(l_1 + l_2)
```

```
## [1, 'pomme', 5, 7, 9, 11]
```

Idem avec des n-uplets =

```
t_1 = (1, "pomme", 5, 7)
t_2 = (9, 11)
print(t_1 + t_2)
```

```
## (1, 'pomme', 5, 7, 9, 11)
```

En “multipliant” une liste par un scalaire  $n$ , Python répète  $n$  fois cette liste :

```
print(3*l_1)
```

```
## [1, 'pomme', 5, 7, 1, 'pomme', 5, 7, 1, 'pomme', 5, 7]
```

Idem avec des n-uplets :

```
print(3*t_1)
```

```
## (1, 'pomme', 5, 7, 1, 'pomme', 5, 7, 1, 'pomme', 5, 7)
```

## 4.2 Opérateurs de comparaison

Les opérateurs de comparaisons permettent de comparer entre eux des objets de tous les types de base. Le résultat d'un test de comparaison produit des valeurs booléennes.

TABLE 4.1 – Opérateurs de comparaison

Opérateur	Opérateur en Python	Description
<code>=</code>	<code>==</code>	Égal à

Opérateur	Opérateur en Python	Description
$\neq$	<code>!=</code> (ou <code>&lt;&gt;</code> )	Différent de
$>$	<code>&gt;</code>	Supérieur à
$\geq$	<code>&gt;=</code>	& Supérieur ou égal à
$<$	<code>&lt;</code>	Inférieur à
$\leq$	<code>&lt;=</code>	Inférieur ou égal à
$\in$	<code>in</code>	Dans
$\notin$	<code>not in</code>	Exclu

### 4.2.1 Égalité, inégalité

Pour tester l'égalité de contenu entre deux objets :

```
a = "Hello"
b = "World"
c = "World"
print(a == c)
```

```
## False
```

```
print(b == c)
```

```
## True
```

L'inégalité entre deux objets :

```
x = [1,2,3]
y = [1,2,3]
z = [1,3,4]
print(x != y)
```

```
## False
```

```
print(x != z)
```

```
## True
```

### 4.2.2 Infériorité et supériorité, stricts ou larges

Pour savoir si un objet est inférieur (strictement ou non) ou supérieur (strictement ou non) à un autre :

```
x = 1
y = 1
z = 2
print(x < y)
```

```
## False
```

```
print(x <= y)
```

```
## True
```

```
print(x > z)
```

```
## False
```

```
print(x >= z)
```

```
## False
```

On peut également effectuer la comparaison entre deux chaînes de caractères. La comparaison s'effectue en fonction de l'ordre lexicographique :

```
m_1 = "mange"
m_2 = "manger"
m_3 = "boire"
print(m_1 < m_2) # mange avant manger
```

```
## True
```

```
print(m_3 > m_1) # boire avant manger
```

```
## False
```

Lorsque l'on compare deux listes entre-elles, Python fonctionne pas à pas. Regardons à travers un exemple comment cette comparaison est effectuée.

Créons deux listes :



```
x = [1, 3, 5, 7]
y = [9, 11]
```

Python va commencer par comparer les premiers éléments de chaque liste (ici, c'est possible, les deux éléments sont comparables ; dans le cas contraire, une erreur serait retournée) :

```
print(x < y)
```

```
## True
```

Comme  $1 < 9$ , Python retourne `True`.

Changeons `x` pour que le premier élément soit supérieur au premier de `y`

```
x = [10, 3, 5, 7]
y = [9, 11]
print(x < y)
```

```
## False
```

Cette fois, comme  $10 > 9$ , Python retourne `False`.

Changeons à présent le premier élément de `x` pour qu'ils soit égal à celui de `y` :

```
x = [10, 3, 5, 7]
y = [10, 11]
print(x < y)
```

```
## True
```

Cette fois, Python compare le premier élément de `x` avec celui de `y`, comme les deux sont identiques, les seconds éléments sont comparés. On peut s'en convaincre en évaluant le code suivant :

```
x = [10, 12, 5, 7]
y = [10, 11]
print(x < y)
```

```
## False
```

### 4.2.3 Inclusion et exclusion

Comme rencontré plusieurs fois dans le Chapitre 3, les tests d'inclusions s'effectuent à l'aide de l'opérateur `in`.

```
print(3 in [1,2, 3])
```

```
## True
```

Pour tester si un élément est exclu d’une liste, d’un n-uplet, dictionnaire, etc., on utilise `not in` :

```
print(4 not in [1,2, 3])
```

```
## True
```

```
print(4 not in [1,2, 3, 4])
```

```
## False
```

Avec un dictionnaire :

```
dictionnaire = {"nom": "Rockwell", "prenom": "Criquette"}  
"age" not in dictionnaire.keys()
```

## 4.3 Opérateurs logiques

Les opérateurs logiques opèrent sur un ou plusieurs objets de type logique (des booléens).

### 4.3.1 Et logique

L’opérateur `and` permet d’effectuer des comparaisons “ET” logiques. On compare deux objets, `x` et `y` (ces objets peuvent résulter d’une comparaison préalable, il suffit juste que tous deux soient des booléens).

Si l’un des deux objets `x` et `y` est vrai, la comparaison “ET” logique retourne vrai :

```
x = True  
y = True  
print(x and y)
```

```
## True
```

Si au moins l’un des deux est faux, la comparaison “ET” logique retourne faux :

```
x = True
y = False
print(x and y)
```

```
## False
```

```
print(y and y)
```

```
## False
```

Si un des deux objets comparés vaut la valeur vide (`None`), alors la comparaison “ET” logique retourne :

- la valeur `None` si l’autre objet vaut `True` ou `None` ;
- la valeur `False` si l’autre objet vaut `False`

```
x = True
y = False
z = None
print(x and z)
```

```
## None
```

```
print(y and z)
```

```
## False
```

```
print(z and z)
```

```
## None
```

### 4.3.2 Ou logique

L’opérateur `or` permet d’effectuer des comparaisons “OU” logiques. À nouveau, on compare deux booléens, `x` et `y`.

Si au moins un des deux objets `x` et `y` est vrai, la comparaison “OU” logique retourne vrai :

```
x = True
y = False
print(x or y)
```

```
## True
```

Si les deux sont faux, la comparaison “OU” logique retourne faux :

```
x = False
y = False
print(x or y)
```

```
## False
```

Si l’un des deux objets vaut `None`, la comparaison “OU” logique retourne :

- `True` si l’autre objet vaut `True` ;
- `None` si l’autre objet vaut `False` ou `None`

```
x = True
y = False
z = None
print(x or z)
```

```
## True
```

```
print(y or z)
```

```
## None
```

```
print(z or z)
```

```
## None
```

### 4.3.3 Non logique

L’opérateur `not`, lorsqu’appliqué à un booléen, évalue ce dernier à sa valeur opposée :

```
x = True
y = False
print(not x)
```

```
## False
```

```
print(not y)
```

```
## True
```

Lorsque l'on utilise l'opérateur `not` sur une valeur vide (`None`), Python retourne `True` :

```
x = None
not x
```

## 4.4 Quelques fonctions

Python dispose de nombreuses fonctions utiles pour manipuler les structures et données. Le tableau suivant en répertorie quelques-unes. Certaines nécessitent le chargement de la librairie `math`, d'autres la librairie `statistics`. Nous verrons d'autres fonctions propres à la librairie NumPy au Chapitre 9.

TABLE 4.2 – Quelques fonctions numériques

Fonction	Description
<code>math.ceil(x)</code>	Plus petits entier supérieur ou égal à <code>x</code>
<code>math.copysign(x, y)</code>	Valeur absolue de <code>x</code> mais avec le signe de <code>y</code>
<code>math.floor(x)</code>	Plus petits entier inférieur ou égal à <code>x</code>
<code>math.round(x, ndigits)</code>	Arrondi de <code>x</code> à <code>ndigits</code> décimales près
<code>math.fabs(x)</code>	Valeur absolue de <code>x</code>
<code>math.exp(x)</code>	Exponentielle de <code>x</code>
<code>math.log(x)</code>	Logarithme naturel de <code>x</code> (en base <code>e</code> )
<code>math.log(x, b)</code>	Logarithme en base <code>b</code> de <code>x</code>
<code>math.log10(x)</code>	Logarithme en base 10 de <code>x</code>
<code>math.pow(x, y)</code>	<code>x</code> élevé à la puissance <code>y</code>
<code>math.sqrt(x)</code>	Racine carrée de <code>x</code>
<code>math.fsum()</code>	Somme des valeurs de <code>x</code>
<code>math.sin(x)</code>	Sinus de <code>x</code>
<code>math.cos(x)</code>	Cosinus de <code>x</code>
<code>math.tan(x)</code>	Tangente de <code>x</code>
<code>math.asin(x)</code>	Arc-sinus de <code>x</code>
<code>math.acos(x)</code>	Arc-cosinus de <code>x</code>
<code>math.atan(x)</code>	Arc-tangente de <code>x</code>
<code>math.sinh(x)</code>	Sinus hyperbolique de <code>x</code>
<code>math.cosh(x)</code>	Cosinus hyperbolique de <code>x</code>
<code>math.tanh(x)</code>	Tangente hyperbolique de <code>x</code>

Fonction	Description
<code>math.asinh(x)</code>	Arc-sinus hyperbolique de <code>x</code>
<code>math.acosh(x)</code>	Arc-cosinus hyperbolique de <code>x</code>
<code>math.atanh(x)</code>	Arc-tangente hyperbolique de <code>x</code>
<code>math.degree(x)</code>	Conversion de <code>x</code> de radians en degrés
<code>math.radians(x)</code>	Conversion de <code>x</code> de degrés en radians
<code>math.factorial()</code>	Factorielle de <code>x</code>
<code>math.gcd(x, y)</code>	Plus grand commun diviseur de <code>x</code> et <code>y</code>
<code>math.isclose(x, y, rel_tol=1e-09, abs_tol=0.0)</code>	Compare <code>x</code> et <code>y</code> et retourne s'ils sont proches au reard de la tolérance <code>rel_tol</code> ( <code>abs_tol</code> est la tolérance minimum absolue)
<code>math.isfinite(x)</code>	Retourne <code>True</code> si <code>x</code> est soit l'infini, soit <code>NaN</code>
<code>math.isinf(x)</code>	Retourne <code>True</code> si <code>x</code> est l'infini, <code>False</code> sinon
<code>math.isnan(x)</code>	Retourne <code>True</code> si <code>x</code> est <code>NaN</code> , <code>False</code> sinon
<code>statistics.mean(x)</code>	Moyenne de <code>x</code>
<code>statistics.median(x)</code>	Médiane de <code>x</code>
<code>statistics.mode(x)</code>	Mode de <code>x</code>
<code>statistics.stdev(x)</code>	Écart-type de <code>x</code>
<code>statistics.variance(x)</code>	Variance de <code>x</code>

## 4.5 Quelques constantes

La librairie `math` propose quelques constantes :

TABLE 4.3 – Quelques constantes intégrées dans Python

Fonction	Description
<code>math.pi</code>	Le nombre Pi ( $\pi$ )
<code>math.e</code>	La constante $e$
<code>math.tau</code>	La constante $\tau$ , égale à $2\pi$
<code>math.inf</code>	L'infini ( $\infty$ )
<code>-math.inf</code>	Moins l'infini ( $-\infty$ )
<code>math.nan</code>	Nombre à virgule flottante <i>not a number</i>

## 4.6 Exercice

1. Calculer le reste de la division euclidienne de 10 par 3.
2. Afficher le plus grand commun diviseur entre 6209 et 4435.
3. Soient deux objets : `a = 18` et `b = -4`. Tester si :

- $a$  est inférieur à  $b$  strictement,
- $a$  est supérieur ou égal à  $b$ ,
- $a$  est différent de  $b$ .

4. Soit la liste  $x = [1, 1, 2, 3, 5, 8]$ . Regarder si :

- 1 est dans  $x$  ;
- 0 est dans  $x$  ;
- 1 et 0 sont dans  $x$  ;
- 1 ou 0 sont dans  $x$  ;
- 1 ou 0 n'est pas présent dans  $x$ .





# Chapitre 5

## Chargement et sauvegarde de données

Pour explorer des données et/ou réaliser des analyses statistiques ou économétriques, il est important de savoir importer et exporter des données.

Avant toute chose, il convient d'évoquer la notion de répertoire courant (*working directory*). En informatique, le répertoire courant d'un processus désigne un répertoire du système de fichier associé à ce processus.

Lorsqu'on lance Jupyter, une arborescence nous est proposée, et nous navigons à l'intérieur de celle-ci pour créer ou ouvrir un *notebook*. Le répertoire contenant le *notebook* est le répertoire courant. Lorsqu'on indiquera à Python d'importer des données (ou d'exporter des objets), l'origine (ou la destination) sera indiquée **relativement** au répertoire courant, à moins d'avoir recours à des chemins absolus (c'est-à-dire un chemin d'accès à partir de la racine /).

Si on lance un programme Python depuis un terminal, le répertoire courant est le répertoire dans lequel on se trouve dans le terminal au moment de lancer le programme.

Pour afficher dans Python le répertoire courant, on peut utiliser le code suivant :

```
import os
cwd = os.getcwd()
print(cwd)

## /Users/ewengallic/Dropbox/Universite_Aix_Marseille/
##   Magistere_2_Programming_for_big_data/Cours/chapters/python/
##   Python_pour_economistes
```

### Remarque 5.0.1

La fonction `listdir()` de la librairie `os` est très pratique : elle permet de lister tous les documents et répertoires contenus dans le répertoire courant, ou dans n'importe quel répertoire si le paramètre `path` renseigne le chemin (absolu ou relatif). Après avoir importé la fonction (`from os import getcwd`), on peut l'appeler : `os.listdir()`.

## 5.1 Charger des données

En fonction du format d'enregistrement des données, les techniques d'importation de données diffèrent.

### Remarque 5.1.1

Le Chapitre 10 propose d'autres manières d'importer les données, avec la librairie `pandas`.

### 5.1.1 Fichiers textes

Lorsque les données sont présentes dans un fichier texte (ASCII), Python propose d'utiliser la fonction `open()`.

La syntaxe (simplifiée) de la fonction `open()` est la suivante :

```
open(file, mode='r', buffering=-1,
      encoding=None, errors=None, newline=None)
```

Voici à quoi correspondent les paramètres (il en existe d'autres) :

- **file** : une chaîne de caractères indiquant le chemin et le nom du fichier à ouvrir ;
- **mode** : spécifie la manière par laquelle le fichier est ouvert (c.f. juste après pour les valeurs possibles) ;
- **buffering** : spécifie à l'aide d'un entier le comportement à adopter pour la mise en mémoire tampon (1 pour mettre en mémoire par ligne ; un entier > 1 pour indiquer la taille en octets des morceaux à charger en mémoire tampon) ;
- **encoding** : spécifie l'encodage du fichier ;
- **errors** : spécifie la manière de gérer les erreurs d'encodage et de décodage (*e.g.*, **strict** retourne une erreur d'exception, **ignore** permet d'ignorer les erreurs, **replace** de les remplacer, **backslashreplace** de remplacer les données mal formées par des séquences d'échappement) ;
- **newline** : contrôle la fin des lignes (`\n`, `\r`, etc.).

TABLE 5.1 – Valeurs principales pour la manière d'ouvrir les fichiers.

Valeur	Description
<b>r</b>	Ouverture pour lire (défaut)
<b>w</b>	Ouverture pour écrire
<b>x</b>	Ouverture pour créer un document, échoue si le fichier existe déjà
<b>a</b>	Ouverture pour écrire, en venant ajouter à la fin du fichier si celui-ci existe déjà
<b>+</b>	Ouverture pour mise à jour (lecture et écriture)

Valeur	Description
<b>b</b>	À ajouter à un mode d'ouverture pour les fichiers binaires ( <b>rb</b> ou <b>wb</b> )
<b>t</b>	Mode texte (décodage automatique des octets en Unicode). Par défaut si non spécifié (s'ajoute au mode, comme <b>b</b> )

Il est important de bien penser à **fermer le fichier** une fois qu'on a terminé de l'utiliser. Pour ce faire, on utilise la méthode `close()`.

Dans le dossier `fichiers_exemples` se trouve un fichier appelé `fichier_texte.txt` qui contient trois lignes de texte. Ouvrons ce fichier, et utilisons la méthode `.read()` pour afficher son contenu :

```
path = "./fichiers_exemples/fichier_texte.txt"
# Ouverture en mode lecture (par défaut)
mon_fichier = open(path, mode = "r")
print(mon_fichier.read())

## Bonjour, je suis un fichier au format txt.
## Je contient plusieurs lignes, l'idée étant de montrer comment
    fonctionne l'importation d'un tel fichier dans Python.
## Trois lignes devraient suffir.
```

```
mon_fichier.close()
```

Une pratique courante en Python est d'ouvrir un fichier dans un bloc `with`. La raison de ce choix est qu'un fichier ouvert dans un tel bloc est automatiquement refermé à la fin du bloc.

La syntaxe est la suivante :

```
# Ouverture en mode lecture (par défaut)
with open(path, "r") as mon_fichier:
    donnees = fonction_pour_recuperer_donnees_depuis_mon_fichier()
```

Par exemple, pour récupérer chaque ligne comme un élément d'une liste, on peut utiliser une boucle parcourant chaque ligne du fichier. À chaque itération, on récupère la ligne :

```
# Ouverture en mode lecture (par défaut)
with open(path, "r") as mon_fichier:
    donnees = [x for x in mon_fichier]
print(donnees)

## ['Bonjour, je suis un fichier au format txt.\n', 'Je contient
    plusieurs lignes, l'idée étant de montrer comment fonctionne l'
    importation d'un tel fichier dans Python.\n', 'Trois lignes
    devraient suffir.']
```

Note : à chaque itération, on peut appliquer la méthode `strip()`, qui retourne la chaîne de caractère de la ligne, en retirant les éventuels caractères blancs en début de chaîne :

```
# Ouverture en mode lecture (par défaut)
with open(path, "r") as mon_fichier:
    donnees = [x.strip() for x in mon_fichier]
print(donnees)

## ['Bonjour, je suis un fichier au format txt.', "Je contiens
    plusieurs lignes, l'idée étant de montrer comment fonctionne l'
    importation d'un tel fichier dans Python.", 'Trois lignes
    devraient suffir.']
```

On peut également utiliser la méthode `readlines()` pour importer les lignes dans une liste :

```
with open(path, "r") as mon_fichier:
    donnees = mon_fichier.readlines()
print(donnees)

## ['Bonjour, je suis un fichier au format txt.\n', "Je contiens
    plusieurs lignes, l'idée étant de montrer comment fonctionne l'
    importation d'un tel fichier dans Python.\n", 'Trois lignes
    devraient suffir.']
```

Il se peut parfois que l'encodage des caractères pose problème lors de l'importation. Dans ce cas, il peut être une bonne idée de changer la valeur du paramètre `encoding` de la fonction `open()`. Les encodages disponibles sont fonction de la locale. Les valeurs disponibles s'obtiennent à l'aide de la méthode suivante (code non exécuté dans ces notes) :

```
import locale
locale.locale_alias
```

### 5.1.1.1 Importation depuis internet

Pour importer un fichier texte depuis Internet, on peut utiliser des méthodes de la librairie `urllib` :

```
import urllib
from urllib.request import urlopen
url = "http://egalllic.fr/Enseignement/Python/fichiers_exemples/fichier_texte.txt"
with urllib.request.urlopen(url) as mon_fichier:
    donnees = mon_fichier.read()
print(donnees)
```

```
## b"Bonjour, je suis un fichier au format txt.\nJe contiens
    plusieurs lignes, l'id\xc3\xa9e \xc3\xa9tant de montrer comment
    fonctionne l'importation d'un tel fichier dans Python.\nTrois
    lignes devraient suffir."
```

Comme on peut le constater, l'encodage des caractères pose souci ici. On peut appliquer la méthode `decode()` :

```
print(donnees.decode())

## Bonjour, je suis un fichier au format txt.
## Je contiens plusieurs lignes, l'idée étant de montrer comment
    fonctionne l'importation d'un tel fichier dans Python.
## Trois lignes devraient suffir.
```

### 5.1.2 Fichiers CSV

Les fichiers CSV (*comma separated value*) sont très répandus. De nombreuses bases de données exportent leurs données en CSV (e.g., Banque Mondiale, FAO, Eurostat, etc.). Pour les importer dans Python, on peut utiliser le module `csv`.

À nouveau, on utilise la fonction `open()`, avec les paramètres décrits dans la Section 5.1.1. Ensuite, on fait appel à la méthode `reader()` du module `csv` :

```
import csv
with open('./fichiers_exemples/fichier_csv.csv') as mon_fichier:
    mon_fichier_reader = csv.reader(mon_fichier, delimiter=',', quotechar='"')
    donnees = [x for x in mon_fichier_reader]
print(donnees)

## [['nom', 'prénom', 'équipe'], ['Irving', ' "Kyrie"', ' "Celtics"'],
    ['James', ' "Lebron"', ' "Lakers"', ''], ['Curry', ' "Stephen"',
    ' "Golden State Warriors"']]
```

La méthode `reader()` peut prendre plusieurs paramètres, décrits dans le Tableau 5.2.

TABLE 5.2 – Paramètres de la fonction `reader()`

Paramètre	Description
<code>csvfile</code>	L'objet ouvert avec <code>open()</code>
<code>dialect</code>	Paramètre spécifiant le 'dialect' du fichier CSV (e.g., <code>excel</code> , <code>excel-tab</code> , <code>unix</code> )

Paramètre	Description
<code>delimiter</code>	Le caractère délimitant les champs ( <i>i.e.</i> , les valeurs des variables)
<code>quotechar</code>	Caractère utilisé pour entourer les champs contenant des caractères spéciaux
<code>escapechar</code>	Caractère d'échappement
<code>doublequote</code>	Contrôle comment les <i>quotechar</i> apparaissent à l'intérieur d'un champ : quand <code>True</code> , le caractère est doublé, ; quand <code>False</code> , le caractère d'échappement est utilisé en préfixe au <i>quotechar</i>
<code>lineterminator</code>	Chaîne de caractères utilisée pour terminer une ligne
<code>skipinitialspace</code>	Quand <code>True</code> , le caractère blanc situé juste après le caractère de séparation des champs est ignoré
<code>strict</code>	Quand <code>True</code> , retourne une erreur d'exception en cas de mauvais input de CSV

On peut aussi importer un fichier CSV en tant que dictionnaire, à l'aide de la méthode `csv.DictReader()` du module CSV :

```
import csv
chemin = "./fichiers_exemples/fichier_csv.csv"
with open(chemin) as mon_fichier:
    mon_fichier_csv = csv.DictReader(mon_fichier)
    donnees = [ligne for ligne in mon_fichier_csv]
print(donnees)

## [OrderedDict([('nom', 'Irving'), ('prénom', ' "Kyrie"'), ('équipe', ' "Celtics"')]), OrderedDict([('nom', 'James'), ('prénom', ' "Lebron"'), ('équipe', ' "Lakers"'), (None, [''])]), OrderedDict([('nom', 'Curry'), ('prénom', ' "Stephen"'), ('équipe', ' "Golden State Warriors"')])]
```

### 5.1.2.1 Importation depuis internet

Comme pour les fichiers `txt`, on peut charger un fichier CSV hébergé sur Internet :

```
import csv
import urllib.request
import codecs
url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/fichier_csv.csv"
with urllib.request.urlopen(url) as mon_fichier:
    mon_fichier_csv = csv.reader(codecs.iterdecode(mon_fichier, 'utf-8'))
```

```

    donnees = [ligne for ligne in mon_fichier_csv]
print(donnees)

## [['nom', 'prénom', 'équipe'], ['Irving', ' "Kyrie"', ' "Celtics
   "'], ['James', ' "Lebron"', ' "Lakers"', ''], ['Curry', ' "
   Stephen"', ' "Golden State Warriors"']]

```

### 5.1.3 Fichier JSON

Pour importer des fichiers au format JSON (*JavaScript Object Notation*), qui sont très utilisés dès lors qu'on communique avec une API, on peut utiliser la librairie `json`, et sa méthode `load()` :

```

import json
lien = './fichiers_exemples/tweets.json'
with open(lien) as mon_fichier_json:
    data = json.load(mon_fichier_json)

```

Ensuite, on peut afficher le contenu importé à l'aide de la fonction `pprint()` :

```

from pprint import pprint
pprint(data)

## {'created_at': 'Wed Sep 26 07:38:05 +0000 2018',
##  'id': 11,
##  'loc': [{'long': 5.3698}, {'lat': 43.2965}],
##  'text': 'Un tweet !',
##  'user_mentions': [{'id': 111, 'screen_name': 'nom_twittos1'},
##                     {'id': 112, 'screen_name': 'nom_twittos2'}]}

```

#### 5.1.3.1 Importation depuis Internet

Encore une fois, il est possible d'importer des fichiers JSON depuis Internet :

```

import urllib
from urllib.request import urlopen
url = "http://egallic.fr/Enseignement/Python/fichiers_exemples/tweets.json"
with urllib.request.urlopen(url) as mon_fichier:
    donnees = json.load(mon_fichier)
pprint(donnees)

## {'created_at': 'Wed Sep 26 07:38:05 +0000 2018',
##  'id': 11,

```

```
## 'loc': [{'long': 5.3698}, {'lat': 43.2965}],
## 'text': 'Un tweet !',
## 'user_mentions': [{'id': 111, 'screen_name': 'nom_twittos1'},
##                    {'id': 112, 'screen_name': 'nom_twittos2'}]]
```

### 5.1.4 Fichiers Excel

Les fichiers Excel (`xls` ou `xlsx`) sont aussi très largement répandus en économie. Le lecteur est prié de se référer à la Section 10.1.1 pour une méthode d'importation des données Excel avec la librairie `pandas`.

## 5.2 Exporter des données

Il n'est pas rare de devoir exporter ses données, ne serait-ce que pour les partager. À nouveau, la fonction `open()` est mise à contribution, en jouant avec la valeur du paramètre `mode` (c.f. Tableau 5.1).

### 5.2.1 Fichiers textes

Admettons que nous ayons besoin d'exporter des lignes de texte dans un fichier. Avant de donner un exemple avec la fonction `open()`, regardons deux fonctions importantes pour convertir les contenus de certains objets en texte.

La première, `str()`, retourne une version en chaînes de caractères d'un objet. Nous l'avons déjà appliquée à des nombres que l'on désirait concaténer en Section 2.1.4.

```
x = ["pomme", 1, 3]
str(x)
```

Le résultat de cette instruction retourne la liste sous la forme d'une chaîne de caractères : `"['pomme', 1, 3]"`.

La seconde fonction qu'il semble important d'aborder est `repr()`. Cette fonction retourne une chaîne contenant une représentation imprimable à l'écran d'un objet. De plus, cette chaîne peut être lue par l'interprète.

```
y = "Fromage, tu veux du fromage ?\n"
repr(y)
```

Le résultat donne : `"'Fromage, tu veux du fromage ?\n'"`.

Admettons que nous souhaitons exporter deux lignes :



- la première, un texte qui indique un titre (“Caractéristiques de Kyrie Irving”);
- la seconde, un dictionnaire contenant des informations sur Kyrie Irving (c.f. ci-dessous).

Définissons ce dictionnaire :

```
z = { "nom": "Kyrie",
      "prenom": "John",
      "naissance": 1992,
      "equipes": ["Cleveland", "Boston"]}
```

Une des syntaxes pour exporter les données au format `txt` est :

```
# Ouverture en mode lecture (par défaut)
chemin = "chemin/vers/fichier.txt"
with open(chemin, "w") as mon_fichier:
    fonction_pour_exporter()
```

On crée une variable indiquant le chemin vers le fichier. On ouvre ensuite le fichier en mode écriture en précisant le paramètre `mode = "w"`. Puis, il reste à écrire nos lignes dans le fichier.

```
chemin = "./fichiers_exemples/Irving.txt"
with open(chemin, mode = "w") as mon_fichier:
    mon_fichier.write("Caractéristiques de Kyrie Irving\n")
    mon_fichier.writelines(repr(z))
```

Si le fichier est déjà existant, en ayant utilisé `mode="w"`, l’ancien fichier sera écrasé par le nouveau. Si on souhaite ajouter des lignes au fichier existant, on utilisera `mode="a"` par exemple :

```
with open(chemin, mode = "a") as mon_fichier:
    mon_fichier.writelines("\nUne autre ligne\n")
```

Si on souhaite être prévenu si le fichier est déjà existant, et faire échouer l’écriture si tel est le cas, on peut utiliser `mode="x"` :

```
with open(chemin, mode = "x") as mon_fichier:
    mon_fichier.writelines("Une nouvelle ligne qui ne sera pas ajoutée\n")
```

```
## FileNotFoundError: [Errno 17] File exists: './fichiers_exemples/
##   Irving.txt'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

### 5.2.2 Fichiers CSV

En tant qu'économiste, il est plus fréquent d'avoir à exporter les données au format CSV plutôt que texte, du fait de la structure en rectangle des données que l'on manipule. Comme pour l'importation de CSV (c.f. Section 5.1.2), on utilise le module `csv`. Pour écrire dans le fichier, on utilise la méthode `writer()`. Les paramètres de formatage de cette fonction sont les mêmes que ceux de la fonction `reader()` (c.f. Tableau 5.2).

Exemple de création d'un fichier CSV :

```
import csv
chemin = "./fichiers_exemples/fichier_export.csv"
with open(chemin, mode='w') as mon_fichier:
    mon_fichier_ecrire = csv.writer(mon_fichier, delimiter=',',
                                     quotechar='"',
                                     quoting=csv.QUOTE_MINIMAL)
    mon_fichier_ecrire.writerow(['Pays', 'Année', 'Trimestre', 'TC_PIB'])
    mon_fichier_ecrire.writerow(['France', '2017', 'Q4', 0.7])
    mon_fichier_ecrire.writerow(['France', '2018', 'Q1', 0.2])
```

Bien évidemment, la plupart du temps, nous n'écrivons pas à la main chaque entrée. Nous exportons les données contenues dans une structure. La Section 10.1.1 donne des exemples de ce type d'export, lorsque les données sont contenues dans des tableaux à deux dimension créés avec la librairie `Pandas`.

### 5.2.3 Fichier JSON

Il peut être nécessaire de sauvegarder des données structurées au format JSON, par exemple lorsqu'on a fait appel à une API (e.g., l'API de Twitter) qui retourne des objets au format JSON.

Pour ce faire, nous allons utiliser la librairie `json`, et sa méthode `dump()`. Cette méthode permet de sérialiser un objet (par exemple une liste, comme ce que l'on obtient avec l'API Twitter interrogée avec la librairie `twitter-python`) en JSON.

```
import json
x = [1, "pomme", ["pépins", "rouge"]]
y = { "nom": "Kyrie",
      "prenom": "John",
      "naissance": 1992,
      "equipes": ["Cleveland", "Boston"]}
x_json = json.dumps(x)
y_json = json.dumps(y)
print("x_json: ", x_json)
```

```
## x_json:  [1, "pomme", ["p\u00e9pins", "rouge"]]
```

```
print("y_json: ", y_json)
```

```
## y_json:  {"nom": "Kyrie", "prenom": "John", "naissance": 1992, "
    equipes": ["Cleveland", "Boston"]}
```

Comme on peut le constater, on rencontre quelques petits problèmes d’affichage des caractères accentués. On peut préciser, à l’aide du paramètre `ensure_ascii` évalué à `False` que l’on ne désire pas s’assurer que les caractères non-ascii soient échappés par des séquences de type `\uXXXX`.

```
x_json = json.dumps(x, ensure_ascii=False)
y_json = json.dumps(y, ensure_ascii=False)
print("x_json: ", x_json)
```

```
## x_json:  [1, "pomme", ["pépins", "rouge"]]
```

```
print("y_json: ", y_json)
```

```
## y_json:  {"nom": "Kyrie", "prenom": "John", "naissance": 1992, "
    equipes": ["Cleveland", "Boston"]}
```

```
chemin = "./fichiers_exemples/export_json.json"
with open(chemin, 'w') as f:
    json.dump(json.dumps(x, ensure_ascii=False), f)
    f.write('\n')
    json.dump(json.dumps(y, ensure_ascii=False), f)
```

Si on souhaite réimporter dans Python le contenu du fichier `export_json.json` :

```
chemin = "./fichiers_exemples/export_json.json"
with open(chemin, "r") as f:
    data = []
    for line in f:
        data.append(json.loads(line, encoding="utf-8"))
print(data)
```

```
## [['[1, "pomme", ["pépins", "rouge"]]', '{"nom": "Kyrie", "prenom":
    "John", "naissance": 1992, "equipes": ["Cleveland", "Boston"]}']]
```

### 5.2.4 Exercice

1. Créer une liste nommée `a` contenant des informations sur le taux de chômage en France au deuxième trimestre 2018. Cette liste doit contenir trois éléments :
  - l'année ;
  - le trimestre ;
  - la valeur du taux de chômage (9.1%).
2. Exporter au format CSV le contenu de la liste `a`, en le faisant précéder d'une ligne précisant les noms des champs. Utiliser le point virgule comme séparateur de champs.
3. Importer le fichier créé dans la question précédente dans Python.

# Chapitre 6

## Conditions

Souvent, en fonction de l'évaluation d'une expression, on désire réaliser une opération plutôt qu'une autre. Par exemple, lorsqu'on crée une nouvelle variable dans une analyse statistique, et que cette variable prend ses valeurs en fonction d'une autre, on peut être amené à utiliser des **instructions conditionnelles** : “si la valeur est inférieure à  $x$ , alors... sinon, ...”.

Dans ce court chapitre, nous regardons comment rédiger les instructions conditionnelles.

### 6.1 Les instructions conditionnelles `if`

L'instruction conditionnelle la plus simple que l'on peut rencontrer est `if`. Si et seulement si une expression est évaluée à `True`, alors une instruction sera évaluée.

La syntaxe est la suivante :

```
if expression:
    instruction
```

Les lignes après les deux points (`:`) doivent être placées dans un bloc, en utilisant un taquet de tabulation.

#### Remarque 6.1.1

Un bloc de code est un regroupement d'instructions. Des codes imbriqués indentés à la même position font partie du même bloc :

```
ligne du bloc 1
ligne du bloc 1
    ligne du bloc2
    ligne du bloc2
ligne du bloc1
```

Dans le code ci-dessous, nous définissons une variable `x` contenant l'entier 2. L'instruction suivante évalue l'expression `x == 2` (cf. Section @red(#opérateurs-comparaison) pour des rappels sur les opérateurs de comparaison). Si le résultat de cette expression est `Vrai`, alors le contenu du bloc est évalué.

```
x = 2
if x == 2:
    print("Hello")
```

```
## Hello
```

Si on change la valeur de `x` de manière à ce que l'expression `x == 2` retourne `False` :

```
x = 3
if x == 2:
    print("Hello")
```

À l'intérieur du bloc, on peut écrire plusieurs instructions qui seront évaluées si l'expression est `True` :

```
x = 2
if x == 2:
    y = "Hello"
    print(y + ", x vaut : " + str(x))
```

```
## Hello, x vaut : 2
```

#### Remarque 6.1.2

Lorsqu'on rédige son code, il peut-être pratique d'utiliser des instructions conditionnelles `if` pour évaluer ou non certaines parties du code. Par exemple, quand on rédige un script, il arrive des moments où nous devons réévaluer le début, mais que certaines parties ne nécessitent pas d'être réévaluées à chaque fois, comme des sorties graphiques (ce qui prend du temps). Il est possible de commenter ces parties de codes ne nécessitant pas une nouvelle évaluation, ou alors on peut les placer dans un bloc conditionnel :

- au début du script, on crée une variable `graphe = False`;
- avant de créer un graphique, on le place dans un bloc `if graphe`:

Au moment de l'exécution du script, on peut choisir de créer et exporter les graphiques des blocs `if graphe`: en modifiant à sa guise la variable `graphe`.

## 6.2 Les instructions conditionnelles if-else

Si la condition n'est pas vérifiée, on peut proposer des instructions à effectuer, à l'aide des instructions if-else.

La syntaxe est la suivante :

```
if expression:
    instructions
else:
    autres_instruction
```

Par exemple, admettons qu'on veuille créer une variable de chaleur prenant la valeur `chaud` si la valeur de la variable `temperature` dépasse 28 degrés C, `froid` sinon. Admettons que la température est de 26 degrés C :

```
temperature = 26
chaleur = ""
if temperature > 28:
    chaleur = "chaud"
else:
    chaleur = "froid"
print("Il fait " + chaleur)
```

```
## Il fait froid
```

Si la température est à présent de 32 degrés C :

```
temperature = 32
chaleur = ""
if temperature > 28:
    chaleur = "chaud"
else:
    chaleur = "froid"
print("Il fait " + chaleur)
```

```
## Il fait chaud
```

## 6.3 Les instructions conditionnelles if-elif

Si la condition n'est pas vérifiée, on peut en tester une autre et alors évaluer d'autres instructions si cette seconde est vérifiée. Sinon, on peut en tester encore une autre, et ainsi

de suite. On peut aussi proposer des instructions si aucune des conditions n'a été évaluée à `True`. Pour ce faire, on peut utiliser des instructions conditionnelles `if-elif`.

La syntaxe est la suivante :

```
if expression:
    instructions
elif expression_2:
    instructions_2
elif expression_3:
    instructions_3
else:
    autres_instruction
```

L'exemple précédent manque un peu de sens commun. Peut-on dire que lorsqu'il fait 28 degrés C ou moins il fait froid ? Ajoutons quelques nuances :

```
temperature = -4
chaleur = ""
if temperature > 28:
    chaleur = "chaude"
elif temperature <= 28 and temperature > 15:
    chaleur = "tempérée"
elif temperature <= 15 and temperature > 0:
    chaleur = "froide"
else:
    chaleur = "très froide"
print("La température est " + chaleur)
```

```
## La température est très froide
```

#### Remarque 6.3.1

L'avantage d'utiliser des instructions conditionnelles `if-elif` par rapport à écrire plusieurs instructions conditionnelles `if` à la suite est qu'avec la première manière de faire, les comparaisons s'arrêtent dès qu'une est remplie, ce qui est plus efficace.

## 6.4 Exercice

Soit une liste nommée `europa` contenant les valeurs suivantes, sous forme de chaînes de caractères : "Allemagne", "France" et "Espagne".

Soit une seconde liste, nommée `asie`, contenant sous forme de chaînes de caractères : "Vietnam", "Chine" et "Inde".



L'objectif va être de créer une variable `continent` qui va indiquer soit `Europe`, `Asie` ou autre à l'issue de l'exécution du code.

À l'aide d'instructions conditionnelles de type `if-elif`, rédiger un code qui vérifie la valeur d'une variable `pays`, et définit la valeur d'une autre variable nommée `continent` en fonction du contenu observé dans `pays` tel que :

- si la valeur de `pays` est présente dans la liste `europe`, `pays` vaudra `Europe` ;
- si la valeur de `pays` est présente dans la liste `asie`, `pays` vaudra `Asie` ;
- si la valeur de `pays` n'est présente ni dans `europe` ni dans `asie`, la variable `pays` vaudra `Autre`.

Pour ce faire :

1. Créer les deux listes `europe` et `asie` ainsi que la variable `pays` (valant "Espagne") et la variable `continent` (initiée avec une chaîne de caractères vide).
2. Rédiger le code permettant de réaliser l'objectif expliqué, et afficher le contenu de la variable `continent` à l'issue de l'exécution.
3. Changer la valeur de `pays` à `Chine` puis à `Brésil` et dans chacun des cas, exécuter le code rédigé dans la question précédente.



# Chapitre 7

## Boucles

Quand on doit répéter plusieurs fois la même opération, pour un nombre déterminé de fois ou tant qu’une condition est vérifiée (ou tant qu’elle n’est pas vérifiée), on peut utiliser des boucles, ce qui est bien moins pénible que d’évaluer à la main ou à coups de copier/coller la même instruction.

Nous allons aborder deux types de boucles dans ce chapitre :

- celles pour lesquelles nous ne savons pas **a priori** le nombre d’itérations (le nombre de répétitions) à effectuer : les boucles `while()`
- celles pour lesquelles nous savons **a priori** combien d’itérations sont nécessaires : les boucles `for()`

### Remarque 7.0.1

Il est possible d’arrêter une boucle `for()` avant un nombre d’itérations prédéfini ; dans le même esprit, il est possible d’utiliser une boucle `while()` en sachant d’avance le nombre d’itérations à effectuer.

## 7.1 Boucles avec `while()`

Le principe d’une boucle `while()` est que les instructions à l’intérieur de la boucle seront répétées tant qu’une condition est respectée. L’idée est de faire dépendre cette condition d’un ou plusieurs objets qui seront modifiés au cours des itérations (sans cela, la boucle tournerait à l’infini).

La syntaxe est la suivante :

```
while condition:
    instructions
```

Comme pour les instructions conditionnelles (c.f. Section 6), les instructions sont placées à l’intérieur d’un bloc.

Regardons un exemple de boucle `while()` :

```
x = 100
while x/3 > 1:
    print(x/3)
    x = x/3

## 33.333333333333336
## 11.111111111111112
## 3.703703703703704
## 1.234567901234568
```

```
print(x/3>1)
```

```
## False
```

```
print(x/3)
```

```
## 0.41152263374485604
```

Dans cette boucle, à chaque itération, la valeur de `x` divisé par 3 est affichée, puis la valeur de `x` est remplacée par le tiers de sa valeur courante. Cette opération est répétée tant que l'expression `x/3 > 1` retourne `True`.

## 7.2 Boucles avec `for()`

Quand on connaît le nombre d'itérations à l'avance, on pourra utiliser une boucle `for()`. La syntaxe est la suivante :

```
for objet in valeurs_possibles:
    instructions
```

avec `objet` le nom d'une variable locale à la fonction `for()`, `valeurs_possibles` un objet comprenant  $n$  éléments définissant les valeurs que prendra `objet` pour chacun des  $n$  tours, et `instructions` les instructions qui seront exécutées à chaque itération.

Nous allons, dans l'exemple qui suit, calculer le carré des  $n$  premiers entiers. Les valeurs que vont prendre notre variable `objet` (que nous allons appeler `i`) seront les entiers de 1 à  $n$ . Pour obtenir une séquence d'entiers en Python, on peut utiliser la fonction `range()`, qui prend les paramètres suivants :

- `start` : (optionnel, par défaut, 0) valeur de début pour la séquence (inclue) ;
- `stop` : valeur de fin de la séquence (non inclue) ;

— **step** : (optionnel, par défaut 1) le pas.

Avant de calculer la suite des  $n$  premiers carrés, regardons un exemple de fonctionnement de la fonction `range()` :

```
print(list(range(0, 4))) # Les entiers de 0 à 3
```

```
## [0, 1, 2, 3]
```

```
print(list(range(4))) # Les entiers de 0 à 3
```

```
## [0, 1, 2, 3]
```

```
print(list(range(2, 10))) # Les entiers de 2 à 9
```

```
## [2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(list(range(2, 10, 3))) # Les entiers de 2 à 9 par pas de 3
```

```
## [2, 5, 8]
```

Aussi, pour afficher la suite des 10 premiers carrés :

```
n=10
for i in range(0, n+1):
    print("Le carré de %s est %s" % (i,i**2))
```

```
## Le carré de 0 est 0
## Le carré de 1 est 1
## Le carré de 2 est 4
## Le carré de 3 est 9
## Le carré de 4 est 16
## Le carré de 5 est 25
## Le carré de 6 est 36
## Le carré de 7 est 49
## Le carré de 8 est 64
## Le carré de 9 est 81
## Le carré de 10 est 100
```

Lors de la première itération,  $i$  vaut 0. Lors de la seconde,  $i$  vaut 1. Lors de la troisième,  $i$  vaut 2, etc.

Si on veut stocker le résultat dans une liste :

```
n=10
n_entiers_carres = []
for i in range(0, n+1):
    n_entiers_carres.append(i**2)

print(n_entiers_carres)

## [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Il n'est pas obligatoire d'utiliser la fonction `range()` dans une boucle `for()`, on peut définir les valeurs "à la main" :

```
for i in [0, 1, 2, 8, 9, 10]:
    print("Le carré de %s est %s" % (i,i**2))

## Le carré de 0 est 0
## Le carré de 1 est 1
## Le carré de 2 est 4
## Le carré de 8 est 64
## Le carré de 9 est 81
## Le carré de 10 est 100
```

Dans le même esprit, il n'est pas obligatoire d'itérer sur des valeurs numériques :

```
for prenom in ["Pascaline", "Gauthier", "Xuan", "Jimmy"]:
    print("Il y a %s lettre(s) dans le prénom %s" % (len(prenom), prenom))

## Il y a 9 lettre(s) dans le prénom Pascaline
## Il y a 8 lettre(s) dans le prénom Gauthier
## Il y a 4 lettre(s) dans le prénom Xuan
## Il y a 5 lettre(s) dans le prénom Jimmy
```

Rien n'empêche de faire des boucles à l'intérieur de boucles :

```
for i in range(0,3):
    for j in range(0,3):
        print("i vaut %s et j vaut %s" % (i, j))

## i vaut 0 et j vaut 0
## i vaut 0 et j vaut 1
## i vaut 0 et j vaut 2
## i vaut 1 et j vaut 0
## i vaut 1 et j vaut 1
## i vaut 1 et j vaut 2
```

```
## i vaut 2 et j vaut 0
## i vaut 2 et j vaut 1
## i vaut 2 et j vaut 2
```

Comme on peut le constater, l'itération se fait pour chaque valeur de `i`, et pour chacune de ces valeurs, une seconde itération est effectuée sur les valeurs de `j`.

#### Remarque 7.2.1

On utilise souvent les lettres `i` et `j` pour désigner un compteur dans une boucle `for()`, mais ce n'est évidemment pas une obligation.

Dans une boucle, si on désire incrémenter un compteur, on peut utiliser le symbole `+=` plutôt que d'écrire `compteur = compteur + ...` :

```
j = 10
for i in range(0, 4):
    j += 5
    print("Nouvelle valeur pour j : %s" % j)
```

```
## Nouvelle valeur pour j : 15
## Nouvelle valeur pour j : 20
## Nouvelle valeur pour j : 25
## Nouvelle valeur pour j : 30
```

```
print(j)
```

```
## 30
```

## 7.3 Exercice

1. Rédiger un programme très naïf visant à déterminer si un nombre est premier ou non. Pour ce faire :
  1. définir une variable `nombre` contenant un entier naturel de votre choix (pas trop grand),
  2. à l'aide d'une boucle, vérifier si chaque entier jusqu'à la racine carrée de votre nombre, est un diviseur de votre nombre (s'arrêter si jamais c'est le cas)
  3. en sortie de boucle, écrire une instruction conditionnelle indiquant si le nombre est premier ou non.

2. Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera `nombre_mystere`. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera `nb_tirages` contiendra le nombre de tirages réalisés pour obtenir le nombre mystère.

*Note : pour tirer un nombre aléatoirement entre 1 et 100, on peut utiliser la méthode `randint()` du module `random`).*

3. Parcourir les entiers de 1 à 20 à l'aide d'une boucle `for` en affichant dans la console à chaque itération si le nombre courant est pair.
4. Utiliser une boucle `for()` pour reproduire la suite de Fibonacci jusqu'à son dixième terme (la séquence  $F_n$  est définie par la relation de récurrence suivante :  $F_n = F_{n-1} + F_{n-2}$  ; les valeurs initiales sont  $F_0 = 0$  et  $F_1 = 1$ ).



# Chapitre 8

## Fonctions

La plupart du temps, on utilise les fonctions de base ou contenues dans des modules. Cela dit, lorsque l'on récupère des données en ligne ou qu'on doit mettre en forme des données importées depuis diverses sources, il arrive qu'il soit nécessaire de créer ses propres fonctions. L'avantage de créer ses fonctions se révèle dès lors qu'on doit effectuer une suite d'instruction de manière répétée, avec quelques légères différences (on peut alors appliquer les fonctions au sein d'une boucle, comme nous l'avons abordé dans le [Chapitre 7](#)).

### 8.1 Définition

Une fonction est déclarée à l'aide du mot clé **keyword**. Ce qu'elle renvoie est retourné à l'aide du mot clé **return**.

La syntaxe est la suivante :

```
def nom_fonction(parametres):  
    corps_de_la_fonction
```

Une fois que la fonction est définie, on l'appelle en faisant référence à son nom :

```
nom_fonction()
```

Il suffit donc de rajouter des parenthèses au nom de la fonction pour l'appeler. En effet, `nom_fonction` désigne l'objet qui contient la fonction qui est appelée à l'aide de l'expression `nom_fonction()`. Par exemple, si on souhaite définir la fonction qui calcule le carré d'un nombre, voici ce que l'on peut écrire :

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

On peut ensuite l'appeler :

```
print(carre(2))
```

```
## 4
```

```
print(carre(-3))
```

```
## 9
```

### 8.1.1 Ajout d'une description

Il est possible (et fortement recommandé) d'ajouter une description de ce que la fonction fait :

```
def carre(x):
    """retourne le carré de x"""
    return x**2
```

De fait, quand on évalue ensuite l'instruction suivante, la description de la fonction s'affiche :

```
`?`(carre)
```

Dans Jupyter Notebook, après avoir écrit le nom de la fonction, on peut aussi afficher la description en appuyant sur les touches du clavier **Shift** et **Tabulation**.

### 8.1.2 Paramètres d'une fonction

Dans l'exemple de la fonction `carre()` que nous avons créée, nous avons renseigné un seul paramètre, appelé `x`. Si la fonction que l'on souhaite créer nécessite plusieurs paramètres, il faut les séparer par une virgule.

Considérons par exemple le problème suivant. Nous disposons d'une fonction de production  $Y(L, K, M)$ , qui dépend du nombre de travailleurs  $L$  et de la quantité de capital  $K$ , et du matériel  $M$ , telle que  $Y(L, K, M) = L^{0.3}K^{0.5}M^2$ . Cette fonction pourra s'écrire en Python de la manière suivante :

```
def production(l, k, m):
    """
    @param l (float) travail
    @param k (float) capital
    @param m (float) matériel
    @desc Retourne la valeur de la production en fonction
    du travail, du capital et du matériel."""
```

```
"""
return l**0.3 * k**0.5 * m**(0.2)
```

### 8.1.2.1 Appel sans noms de paramètres

En reprenant l'exemple précédent, si on nous donne  $L = 60$  et  $K = 42$  et  $M = 40$ , on peut en déduire la production :

```
prod_val = production(60, 42, 40)
print(prod_val)
```

```
## 46.289449781254994
```

On peut noter que le nom des paramètres n'a pas été mentionné ici. Lors de l'appel de la fonction, la valeur du premier paramètre a été attribué au paramètre défini en premier (`l`), celle du second au second paramètre (`k`) et enfin celle du troisième au troisième paramètre (`m`).

### 8.1.2.2 Paramètres positionnels paramètres par mots-clés

Il existe deux types de paramètres que l'on peut donner à une fonction en Python :

- les paramètres positionnels ;
- les paramètres par mots-clés.

Contrairement aux paramètres positionnels, les paramètres par mot clé ont une valeur attribuée par défaut. On parle de paramètre formel pour désigner les paramètres de la fonction (les variables utilisées dans le corps de la fonction) et de paramètre effectif pour désigner la valeur que l'on souhaite donner au paramètre formel. Pour définir la valeur à donner à un paramètre formel, on utilise le symbol d'égalité. Lors de l'appel de la fonction, si l'utilisateur ne définit pas explicitement une valeur, celle par défaut sera affectée. Ainsi, il n'est pas forcément nécessaire de préciser les paramètres par mots-clés lors de l'appel de la fonction.

Il est important de noter que les arguments positionnels (ceux qui n'ont pas de valeur par défaut) doivent apparaître en premier dans la liste des paramètres.

Prenons un exemple avec deux paramètres positionnels (`l` et `m`) et un paramètre par mot-clé (`k`) :

```
def production_2(l, m, k=42):
    """
    @param l (float) travail
    @param m (float) matériel
    @param k (float) capital (defaut : 42)
```

```
@desc Retourne la valeur de la production en fonction
du travail, du capital et du matériel.
"""
return l**0.3 * k**0.5 * m**(0.2)
```

La fonction `production_2()` peut s'appeler, pour donner le même résultat, des trois manières suivantes :

```
# En nommant tous les paramètres, en omettant k
prod_val_1 = production_2(l = 42, m = 40)
# En nommant tous les paramètres et en précisant k
prod_val_2 = production_2(l = 42, m = 40, k = 42)
# En nommant uniquement le paramètre mot-clé k
prod_val_3 = production_2(42, 40, k = 42)
# En ne nommant aucun paramètre
prod_val_4 = production_2(42, 40, 42)
res = [prod_val_1, prod_val_2, prod_val_3, prod_val_4]
print(res)
```

```
## [41.59215573604822, 41.59215573604822, 41.59215573604822,
    41.59215573604822]
```

#### Remarque 8.1.1

Si la fonction contient plusieurs paramètres positionnels ; lors de l'appel :

- soit on nomme tous les paramètres positionnels par leur nom ;
- soit aucun ;
- il n'y a pas d'entre deux.

Du moment que tous les paramètres positionnels sont nommés lors de l'appel, on peut les faire figurer dans des ordres différents :

```
def production_3(a, l, m = 40, k=42):
    """
    @param a (float) productivité totale des facteurs
    @param l (float) travail
    @param m (float) matériel (défaut : 40)
    @param k (float) capital (défaut : 42)
    @desc Retourne la valeur de la production en fonction
    du travail, du capital et du matériel.
    """
    return a * l**0.3 * k**0.5 * m**(0.2)

prod_val_1 = production_3(1, 42, m = 38)
prod_val_2 = production_3(a = 1, l = 42)
```

```

prod_val_3 = production_3(l = 42, a = 1)
prod_val_4 = production_3(m = 40, l = 42, a = 1)
res = [prod_val_1, prod_val_2, prod_val_3, prod_val_4]
print(res)

## [41.16765711449734, 41.59215573604822, 41.59215573604822,
    41.59215573604822]

```

### 8.1.2.3 Fonction comme paramètre

Une fonction peut être fournie en paramètre à une autre fonction.

```

def carre(x):
    """Retourne le carré de x"""
    return x**2

def appliquer_carre_4(fun):
    """Applique la fonction `fun` à 4"""
    return fun(4)
print(appliquer_carre_4(carre))

## 16

```

## 8.2 Portée

Lorsque une fonction est appelée, le corps de cette fonction est interprété. Les variables ayant été définies dans le corps de la fonction sont assignées à un *namespace* local. C'est-à-dire qu'elles ne vivent qu'à l'intérieur de cet espace local, qui est créé au moment de l'appel de la fonction et détruit à la fin de celui-ci. On parle alors de portée des variables. Ainsi, une variable ayant une portée locale (assignée dans l'espace local) peut avoir le même nom qu'une variable globale (définie dans l'espace de travail global), sans pour autant désigner le même objet, ou écraser cet objet.

Regardons cela à travers un exemple.

```

# Définition d'une variable globale :
valeur = 1
# Définition d'une variable locale à la fonction f
def f(x):
    valeur = 2
    nouvelle_valeur = 3

```

```
print("valeur vaut :", valeur)
print("nouvelle_valeur vaut :", nouvelle_valeur)
return x + valeur
```

Appelons la fonction `f()`, puis regardons la valeur de `valeur` et celle de `nouvelle_valeur` après l'exécution de la fonction.

```
res = f(3)
```

```
## valeur vaut : 2
## nouvelle_valeur vaut : 3
```

```
print("valeur vaut :", valeur)
```

```
## valeur vaut : 1
```

```
print("nouvelle_valeur vaut :", nouvelle_valeur)
```

```
## NameError: name 'nouvelle_valeur' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Comme on peut le constater, durant l'appel, la variable locale du nom `valeur` valait 2. Cette variable ne faisait pas référence à la variable du même nom définie dans l'environnement global. À l'issue de l'exécution de la fonction `f()`, cette variable `valeur` locale est supprimée, et il en est de même pour la variable locale `nouvelle_valeur`, qui n'existe pas dans l'environnement global (d'où l'erreur retournée).

Sans trop rentrer trop dans les détails, il semble important de connaître quelques principes à propos de la portée des variables. Les variables sont définies dans des environnements, qui sont imbriqués les uns dans les autres. Si une variable n'est pas définie dans le corps d'une fonction, Python ira chercher dans un environnement parent.

```
valeur = 1
def f(x):
    return x + valeur
```

```
print(f(2))
```

```
## 3
```

Si on définit une fonction à l'intérieur d'une autre fonction, et qu'on appelle une variable non définie dans le corps de cette fonction, Python ira chercher dans l'environnement directement

supérieur. S'il ne trouve pas, il ira chercher dans l'environnement encore supérieur, et ainsi de suite jusqu'à l'environnement global.

```
# La variable valeur n'est pas définie dans
# l'environnement local de g().
# Python va alors chercher dans f().
valeur = 1
def f():
    valeur = 2
    def g(x):
        return x + valeur
    return g(2)
print(f())
```

```
## 4
```

```
# La variable valeur n'est définie ni dans g() ni dans f()
# mais dans l'environnement supérieur (ici, global)
valeur = 1
def f():
    def g(x):
        return x + valeur
    return g(2)
print(f())
```

```
## 3
```

Si on définit une variable dans le corps d'une fonction et que l'on souhaite qu'elle soit accessible dans l'environnement global, on peut utiliser le mot-clé `global` :

```
def f(x):
    global y
    y = x+1
f(3)
print(y)
```

```
## 4
```

#### Remarque 8.2.1

La variable que l'on souhaite définir de manière globale depuis un espace local de la fonction ne doit pas avoir le même nom d'un des paramètres.

## 8.3 Fonctions lambda

Python propose ce que l'on appelle des fonctions lambdas, ou encore des fonctions anonymes. Une fonction lambda ne possède qu'une seule instruction dont le résultat est celui de la fonction.

On les définit à l'aide du mot-clé `lambda`. La syntaxe est la suivante :

```
nom_fonction = lambda parametres : retour
```

Les paramètres sont à séparer par des virgules.

Reprenons la fonction `carre()` créée précédemment :

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

La fonction lambda équivalent s'écrit :

```
carre_2 = lambda x: x**2
print(carre_2(4))
```

```
## 16
```

Avec plusieurs paramètres, regardons la fonction lambda équivalente à la fonction `production()` :

```
def production(l, k, m):
    """
    @param l (float) travail
    @param k (float) capital
    @param m (float) matériel
    @desc Retourne la valeur de la production en fonction
    du travail, du capital et du matériel.
    """
    return l**0.3 * k**0.5 * m**(0.2)
```

```
production_2 = lambda l,k,m : l**0.3 * k**0.5 * m**(0.2)
print(production(42, 40, 42))
```

```
## 40.987803063838406
```

```
print(production_2(42, 40, 42))
```

```
## 40.987803063838406
```



## 8.4 Retour de plusieurs valeurs

Il peut parfois être pratique de retourner plusieurs éléments en retour d'une fonction. Bien que la liste se porte candidate à cette fonctionnalité, il peut-être plus avisé d'utiliser un dictionnaire, pour pouvoir accéder aux valeurs grâce à leur clé!

```
import statistics
def stat_des(x):
    """Retourne la moyenne et l'écart-type de `x`"""
    return {"moyenne": statistics.mean(x),
            "ecart_type": statistics.stdev(x)}
x = [1,3,2,6,4,1,8,9,3,2]
res = stat_des(x)
print(res)

## {'moyenne': 3.9, 'ecart_type': 2.8460498941515415}

print("La moyenne vaut %s et l'écart-type vaut %s" %
      (res["moyenne"], res["ecart_type"]))

## La moyenne vaut 3.9 et l'écart-type vaut 2.8460498941515415
```

## 8.5 Exercice

1. Créer une fonction nommée `somme_n_entiers` qui retourne la somme des  $n$  premiers entiers. Son seul paramètre sera  $n$ .
2. À l'aide d'une boucle, afficher la somme des 2 premiers entiers, puis 3 premiers entiers, puis 4 premiers entiers, etc. jusqu'à 10.
3. Créer une fonction qui à partir de deux points représentés par des couples de coordonnées  $(x_1, y_1)$  et  $(x_2, y_2)$  retourne la distance euclidienne entre ces deux points. Proposer une seconde solution à l'aide d'une fonction `lambda`.



# Chapitre 9

## Introduction à Numpy



# Chapitre 10

## Manipulation de données avec Pandas

### 10.1 Importation et exportation de données

#### 10.1.1 Fichiers Excel

### 10.2 Sélection

### 10.3 Filtrage

### 10.4 Retrait des valeurs dupliquées

### 10.5 Modification des colonnes

### 10.6 Tri

### 10.7 Jointures

### 10.8 Agrégation

### 10.9 Stacking et unstacking



# Chapitre 11

## Visualisation de données





# Chapitre 12

## Programmation parallèle



# Chapitre 13

## References

Briggs, Jason R. 2013. *Python for Kids : A Playful Introduction to Programming*. no starch press.

Grus, Joel. 2015. *Data Science from Scratch : First Principles with Python*. “ O’Reilly Media, Inc.”

McKinney, Wes. 2017. *Python for Data Analysis : Data Wrangling with Pandas, Numpy, and Ipython (2nd Edition)*. “ O’Reilly Media, Inc.”

Navaro, Pierre. 2018. “Python Notebooks.” <https://github.com/pnavaro/python-notebooks>.

VanderPlas, Jake. 2016. *Python Data Science Handbook : Essential Tools for Working with Data*. “ O’Reilly Media, Inc.”