Python pour les économistes

Ewen Gallic

Octobre 2018

Contents

List of Tables

List of Figures

Propos liminaires

```
> 1 + 1
```

```
## [1] 2
```

Ici, on va parler de ce qu'est un algorithme, et pourquoi c'est cool de savoir en rédiger.

Chapter 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 corresponderont à cette version, non pas aux précédentes.

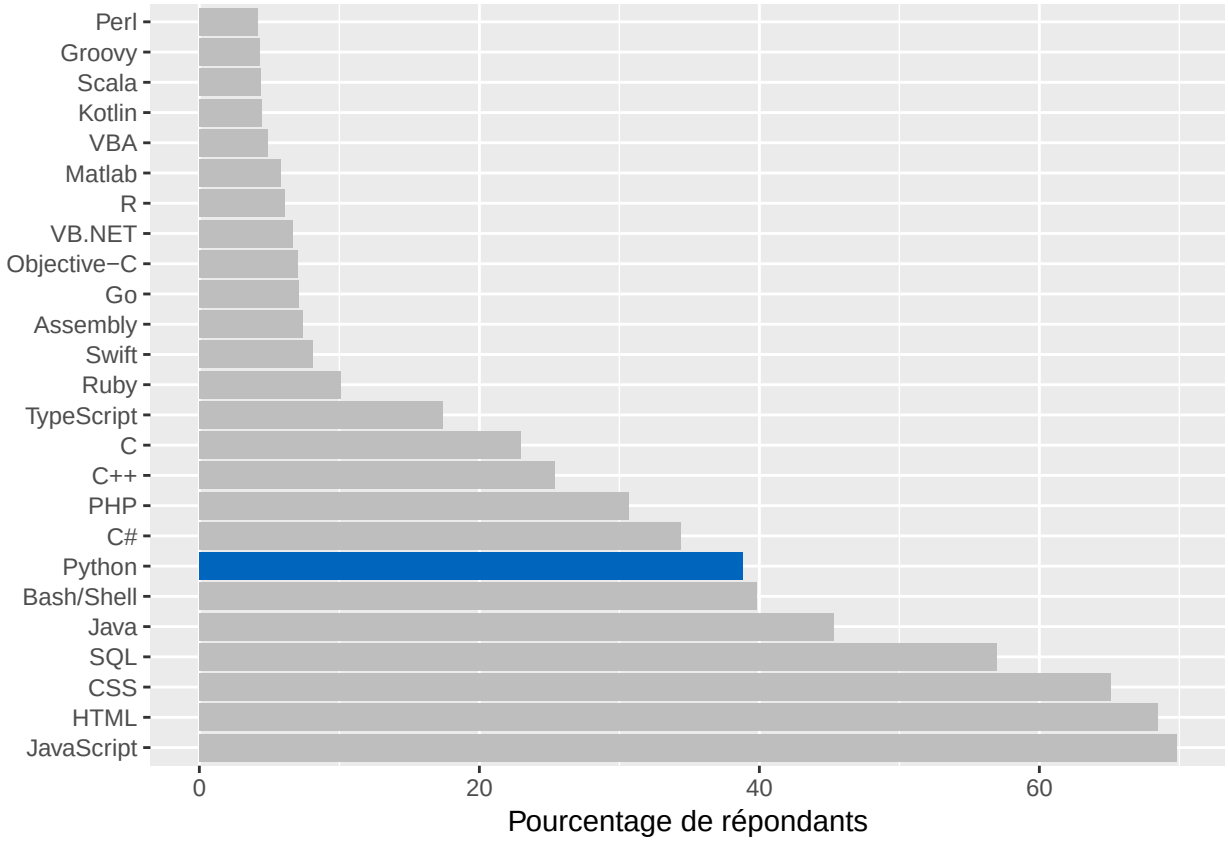


Figure 1.1 – Langages de programmation, de scripting et de balisage.

Comparativement à la version 2.7, la version 3.0 a apporté des modofications 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 :

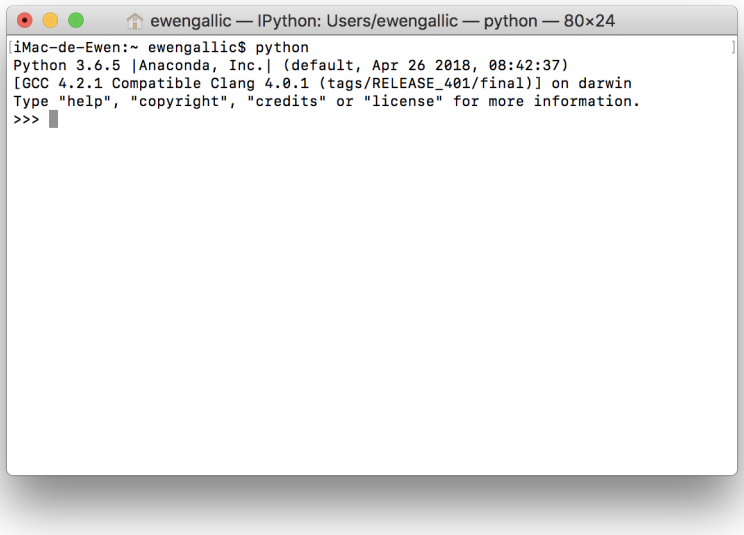


Figure 1.2 – Python dans un terminal.

On note la présence des caractères `>>>` (*prompt*), qui invitent l'utilisateur à inscrie 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 davantages 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.

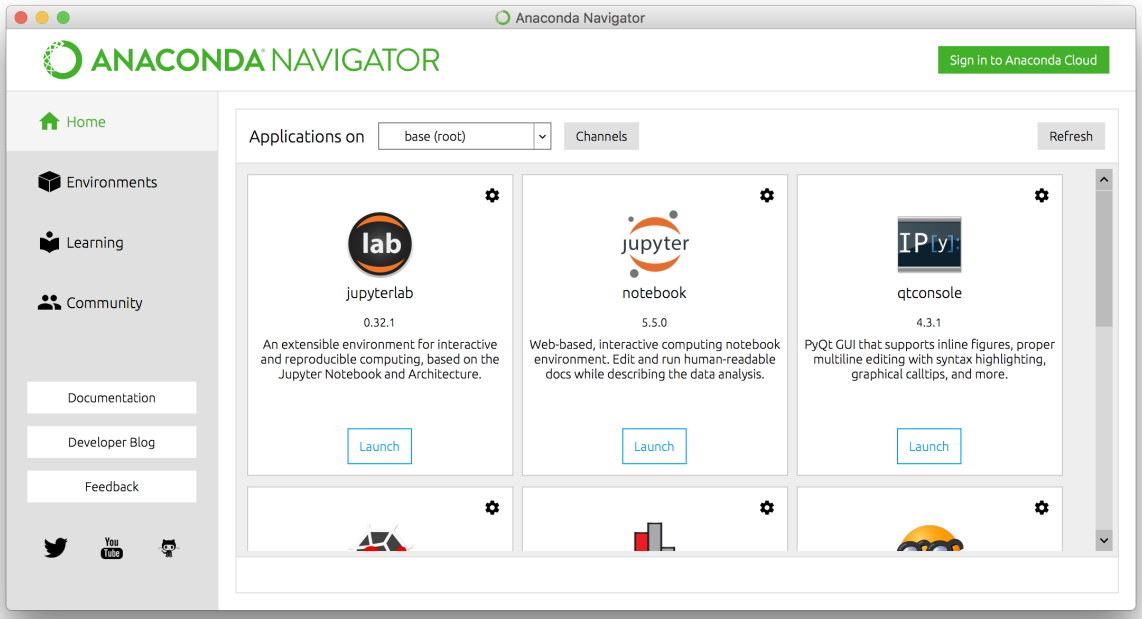


Figure 1.3 – Fenêtre d'accueil d'Anaconda.

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

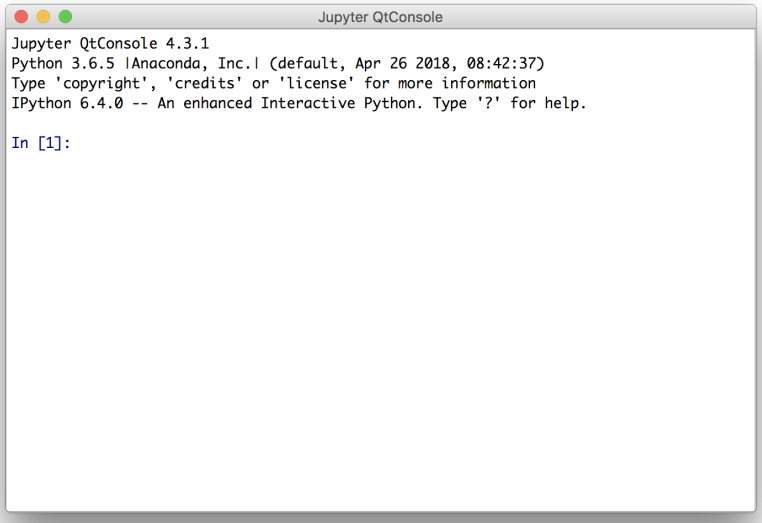


Figure 1.4 – Console IPython.

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 débugger, 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.

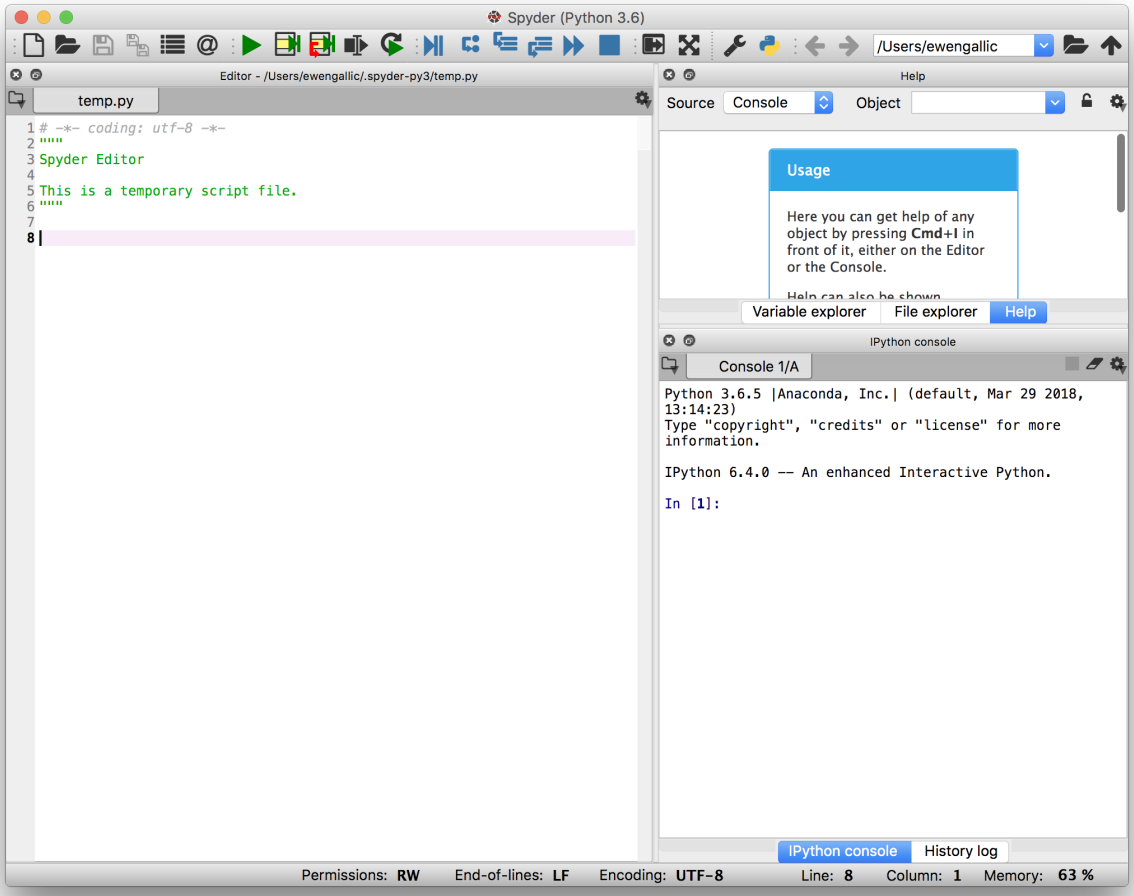


Figure 1.5 – Spyder.

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 en 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?.

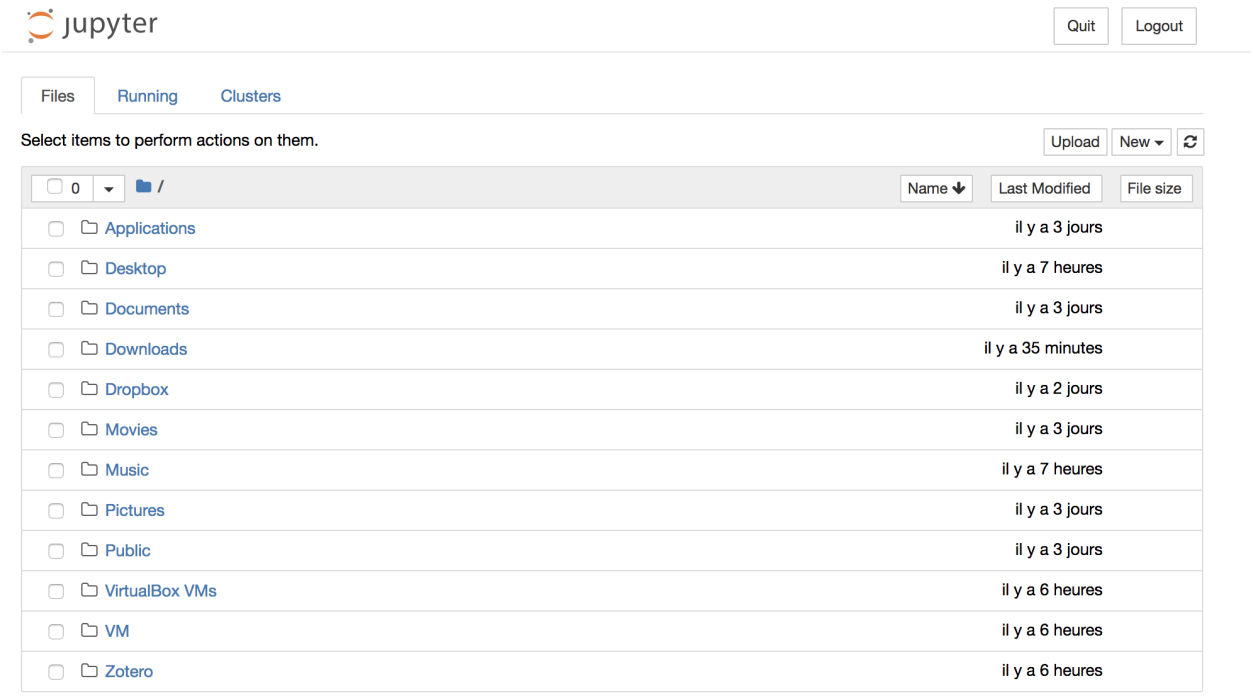


Figure 1.6 – Jupyter.

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.

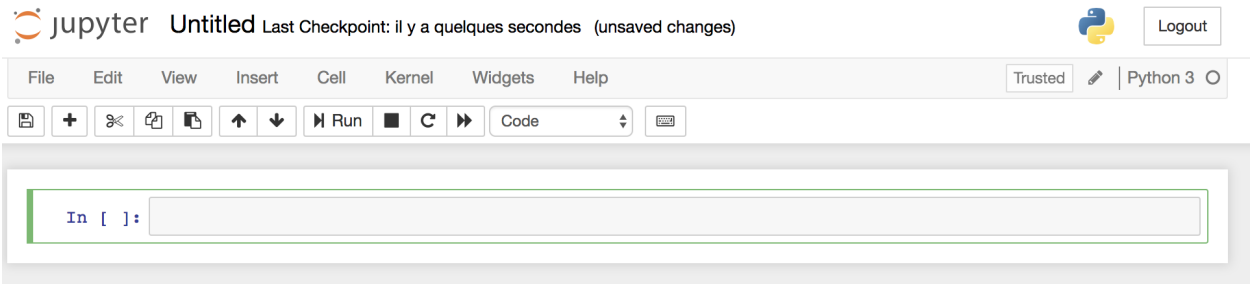


Figure 1.7 – Un notebook vide.

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`.

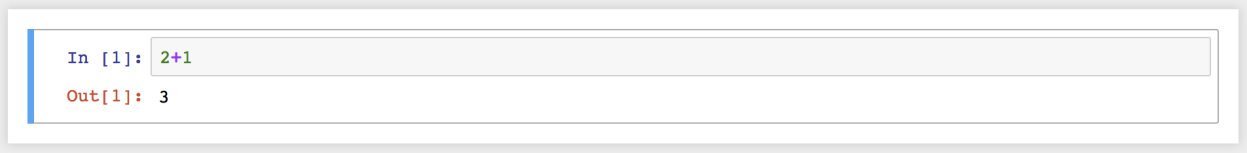


Figure 1.8 – Cellule évaluée.

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`.

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 Assignation 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` renvoit 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 accroitre la lisibilité du nom des variables, plusieurs méthodes existes. 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
```

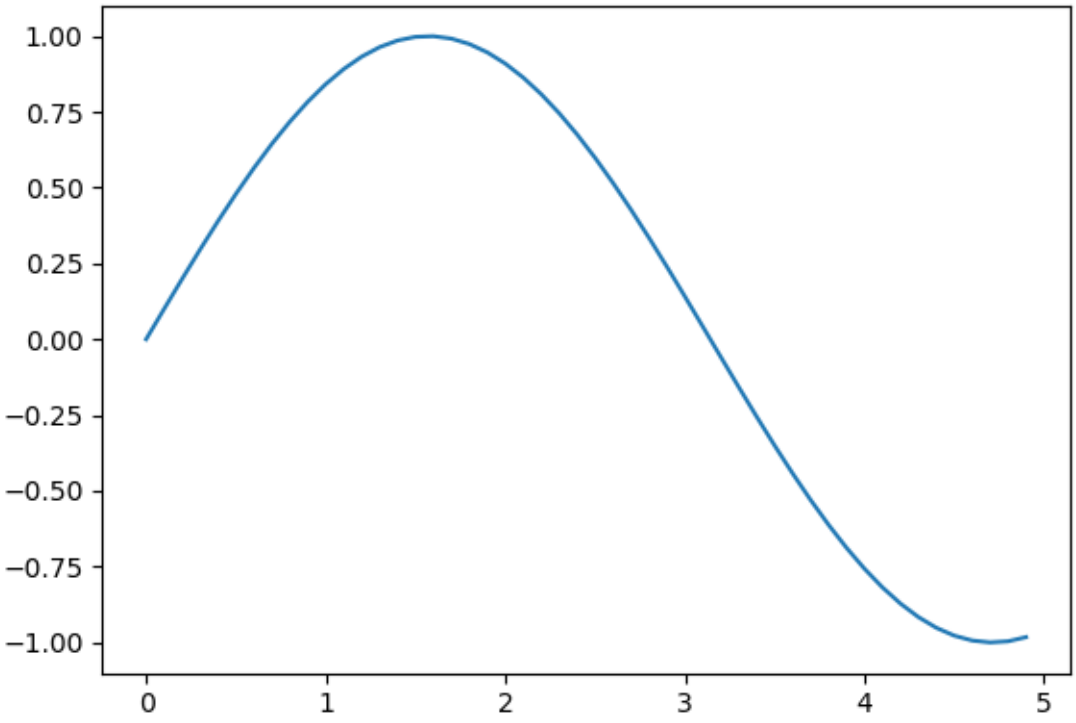
Cette pratique est courante pour abréger les noms des modules que l'on va être amené à utiliser beaucoup. Par exemple, pour `pandas`, il est coutume d'écourter le nom en `pd` :

```
> 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 complèxe, le bon réflèxe 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.

Chapter 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 bouléens (`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 caracrère 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
\n	Nouvelle ligne	\r	Retour à la ligne
\t	Tabulation	\b	Retour arrière
\	Barre oblique inversée	\'	Apostrophe
\"	Apostrophe double	\`	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 ie 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 veleur 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.
+ - Noooooooon ! Noooon !"'''
+ 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
   .", '- Noooooooon ! Noooon !"']
```

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

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 $3,2^{1}2$, on procèdera comme suit :

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

```
## 3200000000000.0
```

```
> print(y)
```

```
## 3200000000000.0
```

2.2.3 Nombres complèxes

Python permet nativement de manipuler des nombres complèxes, 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 complèxes. Par exemple, pour accéder au conjugué, Python fournit la méthode `conjugate()` :

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

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

L'accès à la partie réelle d'un complèxe 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 complèxe

La conversion d'un nombre ou d'une chaîne de caractères en nombre complèxe 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 moduels 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 `dates`, `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 ddes 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'anné (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 er 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 (convientionn 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 renseignant 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 renseignant 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())
```

```
## 2018-10-07 19:31:35.988621
```

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

```
## 2018-10-07 19:31:35.990870
```

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 loins, notamment en ce qui concerne la gestion des fuseaux horaires. Cette librarie 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).

Chapter 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'indiçage :

```
> 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 découpement, 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).

```
> print(x[0])
```

```
## 1
```

3.2.2 Modification

Contrairement aux listes, les n-uplets sont **inaltérables** (c'est-à-dire ne pouvant pas être modifés 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', 'Marseille', 'Aix-en-Provence', 'Rennes'}
```

De manière équivalent, 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', 'Marseille', 'Aix-en-Provence', 'Rennes'}
```

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', 'Marseille', 'Aix-en-Provence', 'Rennes'}
```

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)
```

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

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

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

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

3.3.1.2 Suppression

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

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

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

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)
```

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

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', 'Marseille', 'Aix-en-Provence', 'Toulon'}
```

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

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

3.3.4 Conversion en liste

Un des intérêts des ensembles est 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 entrée, 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(#stucture-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'appliation 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`.

Chapter 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 flotante (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 données, 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 2×2, 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 appliquées 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 symble + 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
=	==	Égal à

Opérateur	Opérateur en Python	Description
$\neq$	!= (ou <>)	Différent de
$>$	>	Supérieur à
$\geq$	>=	& Supérieur ou égal à
$<$	<	Inférieur à
$\leq$	<=	Inférieur ou égal à
$\in$	in	Dans
$\notin$	not in	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 inférieur (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 élement 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`. Nous verrsons d'autres fonctions propres à la librairie `NumPy` au Chapitre 9.

Table 4.2 – Quelques fonctions numériques

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

Fonction	Description
`math.asinh(x)`	Arc-sinus hyperbolique de `x`
`math.acosh(x)`	Arc-cosinus hyperbolique de `x`
`math.atanh(x)`	Arc-tangente hyperbolique de `x`
`math.degree(x)`	Conversion de `x` de radians en degrés
`math.radians(x)`	Conversion de `x` de degrés en radians
`math.factorial()`	Factorielle de `x`
`math.gcd(x, y)`	Plus grand commun diviseur de `x` et `y`
`math.isclose(x, y, rel_tol=1e-09, abs_tol=0.0)`	Compare `x` et `y` et retourne s'ils sont proches au reard de la tolérance `rel_tol` (`abs_tol` est la tolérance minimum absolue)
`math.isfinite(x)`	Retourne `True` si `x` est soit l'infini, soir `NaN`
`math.isinf(x)`	Retourne `True` si `x` est l'infini, `False` sinon
`math.isnan(x)`	Retourne `True` si `x` est `NaN`, `False` sinon

4.5 Quelques constantes

La librairie `math` propose quelques constantes :

Table 4.3 – Quelques constantes intégrées dans Python

Fonction	Description
`math.pi`	Le nombre Pi (π)
`math.e`	La constante e
`math.tau`	La constante τ, égale à 2π
`math.inf`	L'infini (∞)
`-math.inf`	Moins l'infini ($-\infty$)
`math.nan`	Nombre à virgule flotante *not a number*

4.6 Exercice

Remarque 4.6.1

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`.

Chapter 5

Chargement et sauvegarde de données

5.1 Charger des données

5.1.1 Fichiers textes

5.1.2 Fichiers CSV

5.1.3 Fichiers Excel

5.2 Exporter des données

5.2.1 Fichiers textes

5.2.2 Fichiers CSV

Chapter 6

Conditions

6.1 Les conditions `if... else`

6.2 Switch ?

Chapter 7

Boucles

7.1 Boucles avec `while()`

7.2 Boucles avec `for()`

Chapter 8

Fonctions

8.1 Définition

8.2 Portée

8.3 Fonctions lambda

8.4 Erreurs

Chapter 9

Introduction à Numpy

Chapter 10

Manipulation de données avec Pandas

Chapter 11

Visualisation de données

Chapter 12

Programmation parallèle

Chapter 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."