# Programmation Python

## Support de Cours

**Dr Pierre-Jérôme ZOHOU**

# Contents

# Chapter 1

# Initiation à Python

## 1.1 Historique Python

Python est un langage de programmation de haut niveau créé par Guido van Rossum et publié pour la première fois en 1991. Son nom s'inspire de la troupe de comédie britannique Monty Python. Van Rossum a développé Python en réponse à la complexité des langages existants, cherchant à créer un langage qui soit à la fois simple à apprendre et à utiliser.

Python a été conçu avec une syntaxe claire et lisible, ce qui le rend accessible aux débutants tout en étant puissant pour les experts. Il incorpore des éléments de divers langages de programmation tels que ABC, Modula-3, C, et Algol-68, tout en mettant l'accent sur la lisibilité du code.

Les premières versions de Python ont introduit des concepts fondamentaux tels que les types de données dynamiques, la gestion automatique de la mémoire et un large éventail de bibliothèques standard. Python 2.0, sorti en 2000, a apporté des fonctionnalités importantes comme le ramasse-miettes (garbage collection) et le support de l'Unicode.

Python 3.0, publié en 2008, a marqué une rupture importante avec les versions précédentes en introduisant des changements incompatibles pour améliorer la cohérence et la simplicité du langage. Bien que cela ait causé des défis de migration pour les projets existants, Python 3 est rapidement devenu la version de référence grâce à ses améliorations significatives.

Au fil des ans, Python est devenu extrêmement populaire dans divers domaines tels que le développement web, la science des données, l'intelligence

artificielle, le développement de jeux et l'automatisation des tâches. Sa communauté active et ses vastes bibliothèques en font un outil de choix pour de nombreux développeurs et chercheurs.

En résumé, Python est un langage de programmation polyvalent et puissant qui continue d'évoluer grâce à une communauté dynamique et des contributions constantes. Sa simplicité, sa lisibilité et sa flexibilité en font un pilier du développement logiciel moderne.

## 1.2 Quels avantages à utiliser Python ?

Python est un langage moderne avec énormément d'avantages. Au nombre de ceux-ci, nous pouvons citer:

1. **Simplicité et Lisibilité:**
   La syntaxe de Python est claire et intuitive, ce qui facilite l'apprentissage et la compréhension du code. Cela permet aux développeurs de se concentrer sur la résolution de problèmes plutôt que sur les complexités syntaxiques.

2. **Grande Bibliothèque Standard:**
   Python dispose d'une bibliothèque standard riche et extensive, offrant des modules et des packages pour pratiquement toutes les tâches courantes, de la manipulation de fichiers à la gestion des protocoles réseau, ce qui accélère le développement.

3. **Communauté Active et Support:**
   La vaste communauté de Python contribue à une abondance de ressources, de tutoriels, de forums et de bibliothèques tierces. Cela assure un support continu et des mises à jour régulières du langage.

4. **Polyvalence et Applications Multiples:**
   Python est utilisé dans divers domaines, notamment le développement web (avec Django, Flask), la science des données (avec Pandas, NumPy, SciPy), l'intelligence artificielle (avec TensorFlow, PyTorch), l'automatisation des tâches, et bien plus encore.

5. **Interopérabilité et Extensibilité:**
   Python peut facilement s'intégrer avec d'autres langages et technologies. Il peut être utilisé pour écrire des scripts, développer des applications de grande envergure, et même s'intégrer avec des codes en C, C++, ou Java pour des performances optimales.

## 1.3 Quels pré-requis pour coder en Python ?

1. **Connaissances de Base en Informatique**
   Une compréhension de base de l'informatique, comme savoir comment fonctionne un ordinateur, ce que sont les fichiers et les répertoires, et comment utiliser un éditeur de texte, sera utile.

2. **Concepts de Base en Programmation:**
   Une compréhension des concepts fondamentaux de la programmation, tels que les variables, les boucles, les conditions, et les fonctions, est très utile. Si vous êtes débutant complet, il existe de nombreux tutoriels et cours en ligne pour vous aider à apprendre ces concepts en même temps que Python.

3. **Un ordinateur performant:**
   Un ordinateur avec une bonne capacité de calcul est utile pour coder en python. Pour des programmes usuels, un ordinateur moderne de ram 4 pourra faire l'affaire.

4. **Motivation et Curiosité :**
   La programmation peut être complexe et parfois frustrante. Avoir une attitude positive, être curieux, et être prêt à apprendre de ses erreurs est essentiel pour progresser.

## 1.4 Installation de Python

L'installation de Python est un processus relativement simple et intuitif, quel que soit votre système d'exploitation. Nous allons vous proposer de travailler avec un environnement de développement dédié, en l'occurrence ***Spyder***

### 1.4.1 Installation de Spyder sur Windows et Mac

***Spyder*** est disponible pour tous les systèmes d'exploitation. Veuillez vous rendre sur ce site. Défilez vers le bas et téléchargez le setup d'installation qui correspond à votre système d'exploitation si vous possédez Windows ou Mac. Une fois téléchargé, lancez le setup. Le processus d'installation commencera automatiquement. Gardez les paramètres par défaut lors du paramétrage. Une fois l'installation de Spyder terminé, vous pouvez à présent l'ouvrir.

### 1.4.2 Installation de Spyder sur Linux

Pour installer Spyder sur Linux, nous allons utiliser la commande **apt** depuis le terminal.

```
sudo apt-get update

sudo apt-get upgrade

sudo apt install spyder
```

Ceci risque de prendre quelques minutes jusqu'à ce que le téléchargement et l'installation ne se terminent. À la fin, reprenez:

```
sudo apt-get update
sudo apt-get upgrade
```

Et c'est parfait, spyder est prêt à être utilisé sur votre système Linux.

# Chapter 2

# Premiers pas avec Python

## 2.1 Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser en mode interactif, c'est-à-dire de manière à dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

Les trois caractères  supérieur à  constituent le signal d'invite, ou prompt principal, lequel vous indique que Python est prêt à exécuter une commande. Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (Prenez l'habitude d'utiliser votre cahier d'exercices pour noter les résultats qui apparaissent à l'écran) :

Code Python

```
Python 3.12.3 (main, Apr 10 2024, 05:33:47) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 5+3
8
>>> 2 - 9      #les espaces sont optionnels
-7
>>> 7 + 3 * 4    #Hiérarchie des opérations respectée ?
19
>>> (7+3)*4
40
>>> 20/3
6.666666666666667
>>> 20//3
6
>>>
```

On peut déjà tirer quelques conclusions. Les opérations se font comme avec une simple calculatrice, et ici vous pouvez laisser de l'espace entre les chiffres pour aérer vos calculs.

Les règles d'opérations ne changent pas, avec la multiplication qui est prioritaire devant l'addition. On peut aussi obtenir la partie entière d'une division en utilisant // au lieu de /.

Code Python

```
>>> 20.5/3             #Valable
6.833333333333333
>>> 20,5/3             #Erreur
(20, 1.6666666666666667)
```

Veuillez remarquer au passage une règle qui vaut dans tous les langages de programmation : les conventions mathématiques de base sont celles qui sont en vigueur dans les pays anglophones. Le séparateur décimal y est donc toujours un point, et non une virgule comme chez nous. Notez aussi que dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres à virgule flottante (floating point numbers).

## 2.2 Données et Variables

Nous aurons l'occasion de détailler plus loin les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept de grande importance. L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses (tout ce qui est numérisable, en fait), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires. Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de variables de différents types.

Une variable apparaît dans un langage de programmation sous un nom de variable à peu près quelconque (voir ci-après), mais pour l'ordinateur, il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive. À cet emplacement est stockée une valeur bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé.

Cela peut être en fait à peu près n'importe quel objet susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc. Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables (le type entier, le type réel, le type chaîne de caractères, le type liste, etc.). Nous allons expliquer tout cela dans les pages suivantes.

### 2.2.1 Noms de variables et mots réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que altitude, altit ou alt conviennent mieux que x pour exprimer une altitude.

***Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.***

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

1. **Les noms de variables doivent commencer par une lettre ou un underscore :** Les noms de variables peuvent commencer par une lettre **(a-z, A-Z)** ou un underscore, mais pas par un chiffre.

Code Python

```
valid_variable = 10
_valid_variable = 20
1invalid_variable = 30  # Incorrect
```

2. **Les noms de variables peuvent contenir des lettres, des chiffres et des underscores :**
Après le premier caractère, les noms de variables peuvent inclure des lettres, des chiffres et des underscores.

Code Python

```
Variable = 10
variable = 20
```

3. **Les noms de variables ne peuvent pas être des mots réservés :**
Python a des mots réservés qui ne peuvent pas être utilisés comme noms de variables. Voici une liste des mots réservés en Python :

Code Python

```
False, None, True, and, as, assert, async, await,
break, class, continue, def, del, elif, else, except,
finally, for, from, global, if, import, in, is, lambda,
nonlocal, not, or, pass, raise, return, try, while,
with, yield
```

À titre d'illustration, nous avons :

Code Python

```
class = 10  # Incorrect
my_class = 10  # Correct
```

4. **Les noms de variables doivent être significatifs et faciles à lire :**
   Bien que ce ne soit pas une règle syntaxique stricte, il est recommandé de nommer les variables de manière descriptive pour améliorer la lisibilité du code.

Code Python

```
x = 10  # Moins descriptif
number_of_students = 10  # Plus descriptif
```

### 2.2.2 Affectation (Assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons définir une variable et lui affecter une valeur. Les termes affecter une valeur ou assigner une valeur à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe **égale**:

Code Python

```
>>> n = 7              # Variable n créée égal à 7
>>> msg = "Salut"      # affecter la valeur "Salut" à msg
>>> pi = 3.14159       # Variable pi créee
```

Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- trois noms de variables, à savoir **n**, **msg** et **pi** ;
- trois séquences d'octets, où sont encodées le nombre entier **7**, la chaîne de caractères **Salut** et le nombre réel **3,14159**.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un **nom de variable** ;
- lui attribuer un type bien déterminé ;
- créer et mémoriser une valeur particulière ;
- établir un lien entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

### 2.2.3 Afficher la valeur d'une variable

À la suite de l'exercice ci-dessus, nous disposons donc des trois variables n, msg et pi. Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis d'appuyer sur **Entrée**. Python répond en affichant la valeur correspondante :

Code Python

```
>>> n
7
>>> msg
'Salut'
>>> pi
3.14159
>>>
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction **print()**.

Code Python

```
>>> print(msg)
Salut
>>> print(n)
7
>>>
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. La fonction **print()** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type chaîne de caractères .

### 2.2.4 Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit *automatiquement créée* avec le type qui correspond au mieux à la valeur fournie.

Dans l'exercice précédent, par exemple, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent ( **nombre entier pour n, chaîne de caractères pour msg, nombre à virgule flottante (ou float , en anglais) pour pi**). Ceci constitue une particularité intéressante de Python, qui le rattache à une famille particulière de langages où l'on trouve aussi par exemple Lisp, Scheme, et quelques autres. On dira à ce sujet que le typage des variables sous Python est un typage dynamique, par opposition au typage statique qui est de règle par exemple en C++ ou en Java. Dans ces langages, il faut toujours, par des instructions distinctes, d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire figé ).

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé (métaprogrammation, réflexivité), en particulier dans le contexte de la programmation orientée objet (polymorphisme). Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

### 2.2.5 Affectation multiple

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

Code Python

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur :

Code Python

```
>>> a, b = 4, 7.33
>>> a
4
>>> b
7.33
```

**Exercices:**

1. Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

   Code Python

   ```
   >>> largeur = 20
   >>> hauteur = 5 * 9.3
   >>> largeur * hauteur
   930
   ```

2. Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c. Effectuez l'opération a-b//c. Interprétez le résultat obtenu.

### 2.2.6 Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent en les combinant avec des opérateurs pour former des expressions. Exemple :

Code Python

```
>>> a, b = 7.3, 12
>>> y = 3*a + b/5
```

Dans cet exemple, nous commençons par affecter aux variables a et b les valeurs 7,3 et 12. Comme déjà expliqué précédemment, Python assigne automatiquement le type réel à la variable a, et le type entier à la variable b.

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable y le résultat d'une expression qui combine les opérateurs * , + et / avec les opérandes a, b, 3 et 5. Les opérateurs sont les symboles spéciaux utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs.

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, y sera du type réel, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel.

Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Nous avons déjà signalé l'existence de l'opérateur de division entière //. Il faut encore ajouter l'opérateur ** pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc. Nous reparlerons de tout cela au fil des pages suivantes.

Signalons au passage la disponibilité de l'opérateur modulo, représenté par le caractère typographique **pourcentage**. Essayez par exemple :

Code Python

```
>>> 10 % 3          #Prenez note de ce qui se passe.
>>> 10 % 5
```

# Chapter 3

# Structures de Contrôle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence.

## 3.1 La structure If

On peut indiquer à un programme de n'exécuter une instruction (ou une séquence d'instructions) que si une certaine condition est remplie, à l'aide du mot-clé **if** :

Code Python

```
x = 12
if x > 0:
    print("X est positif")
    print("X n'est pas négatif")
```

On a donc comme sortie après compilation:

Code Python

```
X est positif
X n'est pas négatif
```

On remarque ici que la condition est terminée par le symbole :, de plus, la séquence d'instructions à exécuter si la condition est remplie est **indentée**, cela signifie qu'elle est décalée d'un cran (généralement une tabulation ou 4 espaces) vers la droite. Cette indentation est une bonne pratique recommandée quel que soit le langage que vous utilisez, mais en Python, c'est même une **obligation** (sinon, l'interpréteur Python ne saura pas où commence et où se termine la séquence à exécuter sous condition).

Dans certains cas, on souhaite exécuter une série d'instructions si la condition est vérifiée et une autre série d'instructions si elle ne l'est pas. Pour cela, on utilise le mot-clé **else** comme suit:

Code Python

```
x = -1
if x > 0:
    print("X est positif")
    print("X n'est pas négatif")
else:
    print("X est négatif")
```

Le résultat en sortie donne:

Code Python

```
X est négatif
```

Là encore, on remarque que l'indentation est de rigueur pour chacun des deux blocs d'instructions. On note également que le mot-clé **else** se trouve au même niveau que le **if** auquel il se réfère.
Enfin, de manière plus générale, il est possible de définir plusieurs comportements en fonction de plusieurs tests successifs, à l'aide du mot-clé **elif**. **elif** est une contraction de **else if**, qui signifie sinon si.

Code Python

```
x = -1
if x > 0:
    print("X est positif")
    x = 4
elif x > -2:
    print("X est compris entre -2 et 0")
elif x > -4:
    print("X est compris entre -4 et -2")
else:
    print("X est inférieur à -4")
```

Sortie:

Code Python

```
X est compris entre -2 et 0
```

Pour utiliser ces structures conditionnelles, il est important de maîtriser les différents opérateurs de comparaison à votre disposition en Python, dont voici une liste non exhaustive :

ccc

| Opérateur | Comparaison effectuée | Exemple |
|---|---|---|
| < | Plus petit que | `x < 0` |
| > | Plus grand que | `x > 0` |
| <= | Plus petit ou égal à | `x <= 0` |
| >= | Plus grand ou égal à | `x >= 0` |
| == | Égal à | `x == 0` |
| != | Différent de | `x != 0` |

Table 3.1: Opérateurs de comparaison en Python

Il est notamment important de remarquer que, lorsque l'on souhaite tester l'égalité entre deux valeurs, l'opérateur à utiliser est == et non = (qui sert à affecter une valeur à une variable).

## 3.2 Les déclarations Switch

Les déclarations Switch Case, également appelées correspondance de motifs structurels, sont une fonctionnalité courante dans de nombreux langages de programmation. Cependant, elles étaient absentes en Python jusqu'à l'introduction de la version 3.10. Une déclaration Switch Case est un type d'instruction de contrôle qui permet de tester une variable pour savoir si elle est égale à une liste de valeurs. Chaque valeur est appelée un cas, et la variable sur laquelle on effectue la commutation est vérifiée pour chaque cas.

En Python 3.10, la déclaration Switch Case est implémentée en utilisant le mot-clé match, suivi d'une expression. Le mot-clé case est utilisé pour définir différents cas, et le code sous chaque cas est exécuté si l'expression correspond au cas.

### 3.2.1 Pourquoi n'y avait-il pas de déclaration Switch Case en Python avant la version 3.10 ?

Contrairement à d'autres langages de programmation, Python n'incluait pas de déclaration Switch Case avant la version 3.10. La raison principale en est la philosophie de simplicité et de lisibilité de Python. Les créateurs de Python estimaient que l'inclusion de déclarations Switch Case pouvait conduire à un code plus complexe et moins lisible. Au lieu de cela, Python utilisait des instructions if-elif-else pour gérer plusieurs conditions, ce qui servait un objectif similaire aux déclarations Switch Case, mais de manière plus pythonique.

Cependant, avec la sortie de Python 3.10, le langage a introduit des déclarations Switch Case, reconnaissant leur potentiel pour améliorer l'efficacité et la lisibilité du code dans certaines situations.

### 3.2.2 Comment fonctionne le Switch Case en Python 3.10 ?

En Python 3.10, les déclarations Switch Case sont implémentées en utilisant les mots-clés **match** et **case**. Voici un exemple basique de son fonctionnement :

Code Python

```
def switch_case(x):
    match x:
        case 1:
            return "un"
        case 2:
            return "deux"
        default:
            return "inconnu"
```

Dans cet exemple, la fonction `switchcase` prend un argument x. Le mot-clé match est utilisé pour faire correspondre l'argument x à différents cas. Si x est égal à 1, la fonction renvoie "un". Si x est égal à 2, elle renvoie "deux". Le cas default est utilisé comme solution par défaut si x ne correspond à aucun des cas définis.

## 3.3 Structures itératives

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives.

### 3.3.1 Boucle While

En Python, la boucle **WHILE** permet de répéter une instruction plusieurs fois, tant qu'une condition est vraie. Elle permet donc de gagner du temps dans la rédaction de ton code, puisque tu n'écris qu'une seule fois l'instruction qui sera répétée. Mais en plus, elle rend ton programme plus autonome : Python reconnaît lui-même à quel moment il doit sortir de la boucle **WHILE** ! Le rôle d'une boucle **WHILE** est d'exécuter un bloc de code, c'est-à-dire un certain nombre d'instructions, tant qu'une condition est vraie.

La syntaxe est très similaire à celle des structures conditionnelles simples :

Code Python

```
x = 0
while x <= 10:
    print(x)
    x = 2 * x + 2
```

Le code ci-dessus veut dire: Tant que x est inférieur à 10, affiche x et calcule x = 2 * x + 2. Nous avons donc en sortie:

Code Python

```
0
2
6
```

On voit bien ici, en analysant la sortie produite par ces quelques lignes, que le contenu de la boucle est répété plusieurs fois. En pratique, il est répété jusqu'à ce que la variable x prenne une valeur supérieure à 10 (14 dans notre cas). Il faut être très prudent avec ces boucles while car il est tout à fait possible de créer une boucle dont le programme ne sortira jamais, comme dans l'exemple suivant :

Code Python

```
x = 2
y = 0
while x > 0:
    y = y - 1
    print(y)
print("Si on arrive ici, on a fini")
```

En effet, on a ici une boucle qui s'exécutera tant que x est positif, or la valeur de cette variable est initialisée à 2 et n'est pas modifiée au sein de la boucle, la condition sera donc toujours vérifiée et le programme ne sortira jamais de la boucle. Pour information, si vous vous retrouvez dans un tel cas, vous pourrez interrompre l'exécution du programme à l'aide de la combinaison de touches Ctrl + C.

### 3.3.2 Boucle for

La boucle for est une structure de contrôle fondamentale en Python qui permet d'exécuter un bloc de code de manière répétée pour chaque élément d'une séquence.

Si vous avez appris à programmer dans un autre langage que Python, il est possible que vous ayez été habitué(e) à ce que les boucles for soient utilisées pour itérer sur des entiers (par exemple les indices des éléments d'une liste). En Python, le principe de base est légèrement différent : par défaut, on itère sur les éléments d'une liste.

Sachez pour le moment qu'une liste est un ensemble ordonné d'éléments. On peut alors exécuter une série d'instructions pour toutes les valeurs d'une liste :

Code Python

```
for x in [1, 5, 7]:
    print(x)
print("Fin de la boucle")
```

Code Python

```
1
5
7
Fin de la boucle
```

Cette syntaxe revient à définir une variable x qui prendra successivement pour valeur chacune des valeurs de la liste [1, 5, 7] dans l'ordre et à exécuter le code de la boucle (ici, un appel à la fonction print) pour cette valeur de la variable x.

Il est fréquent d'avoir à itérer sur les n premiers entiers. Cela peut se faire à l'aide de la fonction range

Code Python

```
for i in range(1, 10):
    print(i)
```

Ce bloc d'instruction produit le résultat suivant:

Code Python

```
1
2
3
4
5
6
7
8
9
```

La fonction **range(a, b)** retourne un itérable contenant les entiers de a (inclus) à b (exclu) :

Code Python

```
it = range(1, 10)     # it = [1, 2, 3, ..., 9]
it = range(10)        # it = [0, 1, 2, ..., 9]
it = range(0, 10, 2)  # it = [0, 2, 4, ..., 8]
```

On remarque que, si l'on ne donne qu'un argument à la fonction range, l'itérable retourné débute à l'entier 0. Si, au contraire, on passe un troisième argument à la fonction range, cet argument correspond au pas utilisé entre deux éléments successifs.

### 3.3.3 Quelle boucle choisir ?

Lorsque l'on débute la programmation en Python, il peut être difficile de choisir, pour un problème donné, entre les boucles **for** et **while** présentées ici. De manière générale, dès lors que l'on souhaite parcourir les éléments d'une liste ou d'un dictionnaire, on utilisera la boucle **for**. De même, si l'on connait à l'avance le nombre d'itérations que l'on souhaite effectuer, on utilisera une boucle for (couplée avec un appel à la fonction **range**).

Comme son nom l'indique, la boucle **while** sera utilisée dès lors que l'on souhaite répéter une action tant qu" une condition est vérifiée.

### 3.3.4 Exercices

1. Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
   Exemple : 7 14 21 * 28 35 42 * 49 56 63 * 70 77 84 * 91 98 105 * 112 119 126 * 133 140

2. Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme est égal au triple du terme précédent.

3. Écrivez une boucle permettant d'afficher tous les nombres impairs inférieurs à une valeur n initialement fixée.

4. L'utilisateur donne un entier supérieur à 1 et le programme affiche, s'il y en a, tous ses diviseurs propres sans répétition ainsi que leur nombre. S'il n'y en a pas, il indique qu'il est premier. Par exemple :

Code Python

```
Entrez un entier strictement positif : 12
Diviseurs propres sans répétition de 12 : 2 3 4 6 (soit 4
↪  diviseurs propres)
Entrez un entier strictement positif : 13
Diviseurs propres sans répétition de 13 : aucun ! Il est premier
```

# Chapter 4

# Les fonctions

Dans cette nouvelle partie, nous allons étudier une autre notion incontournable de tout langage de programmation qui se respecte : les fonctions. Nous allons notamment définir ce qu'est une fonction et comprendre l'intérêt d'utiliser ces structures puis nous verrons comment créer nos propres fonction en Python ainsi que certains concepts avancés relatifs aux fonctions.

Une fonction est un bloc de code nommé. Une fonction correspond à un ensemble d'instructions créées pour effectuer une tâche précise, regroupées ensemble et qu'on va pouvoir exécuter autant de fois qu'on le souhaite en "l'appelant" avec son nom. Notez "qu'appeler" une fonction signifie exécuter les instructions qu'elle contient.

L'intérêt principal des fonctions se situe dans le fait qu'on va pouvoir appeler une fonction et donc exécuter les instructions qu'elle contient autant de fois qu'on le souhaite, ce qui constitue au final un gain de temps conséquent pour le développement d'un programme et ce qui nous permet de créer un code beaucoup plus clair.

Nous avons déjà vu dans ce qui précède, sans le dire, des fonctions. Par exemple, lorsque l'on écrit :

Code Python

```
>>> print(x)
>>> 7
```

On demande l'appel à une fonction, nommée **print** et prenant un argument (ici, la variable x). La fonction print ne retourne pas de valeur, elle ne fait qu'afficher la valeur contenue dans x sur le terminal. D'autres fonctions, comme **type** retournent une valeur et cette valeur peut être utilisée dans la suite du programme, comme dans l'exemple suivant :

Code Python

```
x = type(1)  # On stocke dans x la valeur retournée par type
y = type(2.)
if x == y:
    print("types identiques")
else:
    print("types différents")
```

Code Python

```
types différents
```

## 4.1 Définition d'une fonction

Pour définir une fonction en Python, on utilise le mot-clé **def** :

Code Python

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2
```

On a ici défini une fonction

- dont le nom est **f** ;
- qui prend un seul argument, noté **x** ;

- qui retourne une valeur, comme indiqué dans la ligne débutant par le mot-clé **return**.

Il est possible, en Python, d'écrire des fonctions retournant plusieurs valeurs. Pour ce faire, ces valeurs seront séparées par des virgules dans l'instruction **return** :

Code Python

```
def f(x):
    y = 5 * x + 2
    z = x + y
    return z // 2, y
```

Enfin, en l'absence d'instruction return, une fonction retournera la valeur **None**.

Il est également possible d'utiliser le nom des arguments de la fonction lors de l'appel, pour ne pas risquer de se tromper dans l'ordre des arguments. Par exemple, si l'on a la fonction suivante :

Code Python

```
def affiche_infos_personne(poids, taille):
    print("Poids: ", poids)
    print("Taille: ", taille)
```

Les trois instructions suivantes s'équivalent:

Code Python

```
affiche_infos_personne(80, 180)
affiche_infos_personne(taille=180, poids=80)
affiche_infos_personne(poids=80, taille=180)
```

Et produisent la même sortie:

Code Python

```
Poids:  80
Taille:  180
```

Notons qu'il est alors possible d'interchanger l'ordre des arguments lors de l'appel d'une fonction si on précise leur nom. Évidemment, pour que cela soit vraiment utile, il est hautement recommandé d'utiliser des noms d'arguments explicites lors de la définition de vos fonctions.

## 4.2 Arguments optionnels d'une fonction

Certains arguments d'une fonction peuvent avoir une valeur par défaut, décidée par la personne qui a écrit la fonction. Dans ce cas, si l'utilisateur ne spécifie pas explicitement de valeur pour ces arguments lors de l'appel à la fonction, c'est la valeur par défaut qui sera utilisée dans la fonction, dans le cas contraire, la valeur spécifiée sera utilisée.

Par exemple, la fonction **print** dispose de plusieurs arguments facultatifs, comme le caractère par lequel terminer l'affichage.

Code Python

```
>>> print("La vie est belle")

>>> La vie est belle
```

Code Python

```
>>> print("La vie est belle", end="--")

>>> La vie est belle--
```

Lorsque vous définissez une fonction, la syntaxe à utiliser pour donner une valeur par défaut à un argument est la suivante :

Code Python

```
def f(x, y=0):  # La valeur par défaut pour y est 0
    return x + 5 * y
```

Attention toutefois, les arguments facultatifs (ie. qui disposent d'une valeur par défaut) doivent impérativement se trouver, dans la liste des arguments, après le dernier argument obligatoire. Ainsi, la définition de fonction suivante n'est pas correcte :

Code Python

```
def f(x, y=0, z):
    return x - 2 * y + z
```

### 4.2.1 Exercices

1. Écrivez une fonction en Python qui prenne en argument une longueur **long** et retourne l'aire du triangle équilatéral de côté **long**.

2.

3. Définissez une fonction surfCercle(R). Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon R en argument. Par exemple, l'exécution de l'instruction : print(surfCercle(2.5)) doit donner le résultat 19.635

4. Définissez une fonction maximum(n1,n2,n3) qui renvoie le plus grand des 3 nombres n1, n2, n3 fournis en arguments. Par exemple, l'exécution de l'instruction : print(maximum(2,5,4)) doit donner le résultat : 5.

5. Définissez une fonction changeCar(ch,ca1,ca2,début,fin) qui remplace tous les caractères ca1 par des caractères ca2 dans la chaîne de caractères ch, à partir de l'indice début et jusqu'à l'indice fin, ces deux derniers arguments pouvant être omis (et dans ce cas, la chaîne est traitée d'une extrémité à l'autre). Exemples de la fonctionnalité attendue :

Code Python

```
>>> phrase = 'Ceci est une toute petite phrase.'
>>> print changeCar(phrase, ' ', '*')
Ceci*est*une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', 8, 12)
Ceci est*une*toute petite phrase.
>>> print changeCar(phrase, ' ', '*', 12)
Ceci est une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', fin = 12)
Ceci*est*une*toute petite phrase.
```

### 4.2.2 Les modules Python

Jusqu'à présent, nous avons utilisé des fonctions (comme **print**) issues de la librairie standard de Python. Celles-ci sont donc chargées par défaut lorsque l'on exécute un script Python. Toutefois, il peut être nécessaire d'avoir accès à d'autres fonctions et/ou variables, définies dans d'autres librairies. Pour cela, il sera utile de charger le module correspondant.

Prenons l'exemple du module math qui propose un certain nombre de fonctions mathématiques usuelles (**sin** pour le calcul du sinus d'un angle, **sqrt** pour la racine carrée d'un nombre, etc.) ainsi que des constantes mathématiques très utiles comme pi. Le code suivant charge le module en mémoire puis fait appel à certaines de ses fonctions et/ou variables :

Code Python

```
import math

print(math.sin(0))

>>> 0.0
```

Code Python

```
print(math.pi)

>>> 3.141592653589793

print(math.cos(2 * math.pi))

>>> 1.0

print(math.sqrt(2))

>>> 1.4142135623730951
```

Vous remarquerez ici que l'instruction d'import du module se trouve nécessairement avant les instructions faisant référence aux fonctions et variables de ce module, faute de quoi ces dernières ne seraient pas définies. De manière générale, vous prendrez la bonne habitude d'écrire les instructions d'import en tout début de vos fichiers Python, pour éviter tout souci.

Enfin, il est possible de renommer le module au moment où vous l'importez, ce qui peut être pratique pour les modules dont le nom est long ou ceux que vous allez utiliser de manière particulièrement fréquente dans votre code. Typiquement, il est de coutume de renommer le module **numpy** en **np** lorsqu'on l'importe :

Code Python

```
import numpy as np

print(np.zeros(5))  #On utilise alors la forme abrégée

>>> [0. 0. 0. 0. 0.]
```

# Chapter 5

# Les Structures de Données

En Python, il existe plusieurs structures de données intégrées qui sont très utiles pour organiser et manipuler les données. Voici un aperçu des structures de données les plus couramment utilisées :

### 5.0.1 Les chaînes de caractère Python

Les chaînes de caractères sont vues par Python comme une collection ordonnée d'éléments. Ceci veut dire que les caractères qui constituent une chaîne sont disposés dans un certain ordre et que l'on peut accéder à chaque caractère à l'aide d'un indice. Attention ! Le premier caractère de la chaîne a pour indice 0 (et non 1).

Code Python

```
>>> chaine = "Algorithmique et Programmation"
>>> chaine[0], chaine[2], chaine[10], chaine[-1]
A g q n
```

Comme vous pouvez le voir, les indices négatifs peuvent être utilisés afin d'accéder aux caractères de la chaîne par la fin. Nous pouvons déterminer la longueur d'une chaîne, c.-à-d. le nombre de ses caractères, à l'aide de la fonction **len()**.

Code Python

```
>>> len(chaine)
30
```

Il est possible de concaténer deux chaînes à l'aide de l'opérateur '+'.

Code Python

```
>>> ch1 = "La raison est la tienne "
>>> ch2 = "mais la chèvre est la mienne"
>>> ch1 + ch2
'La raison est la tienne mais la chèvre est la mienne'
```

**Important!**
*En python, les chaînes de caractères sont des objets immutables, c'est-à-dire qu'il s'agit d'objet que l'on ne peut pas modifier ! En effet, nous verrons que les différents traitements sur les chaines de caractères ne modifient jamais la chaîne de caractères elle-même, elles construisent une nouvelle chaîne de caractères à partir de l'ancienne.*

### Transformer un objet en chaîne de caractères

La fonction **str()** transforme (quand c'est possible) l'objet en chaîne de caractères. Utile pour récupérer, par exemple, tous les chiffres d'un nombre séparément.

Code Python

```
>>> # On récupère le chiffre d'indice 4
>>> nombre = 133675184
>>> str_5e_chiffre = str(nombre)[4]
>>> print(f"Le cinquième chiffre du nombre {nombre} est
↪  {str_5e_chiffre}.")
Le cinquième chiffre du nombre 133675184 est 7.
```

Dans l'exemple, on transforme le nombre ***nombre*** en chaine de caractères ce qui nous permet de récupérer les chiffres un à un sous forme de chaine de caractères. Si on veut les retransformer en nombre pour pouvoir faire des calculs, par exemple, on pourra utiliser int($\mathrm{str}_5 e_c hiffre$)

### Comparaisons entre chaines de caractères

On peut comparer, comme pour les nombres, des chaines de caractères. Le résultat de la comparaison est une booléen : **True** ou **False**. Il peut donc s'utiliser comme condition avec une structure alternative **if** ou répétitive **while**.

Voici les différentes comparaisons possibles :

- chaine1 == chaine2 : Renvoie True si la valeur des deux chaînes sont parfaitement identiques, False sinon.
- chaine1 `!=` chaine2 : Renvoie True si la valeur des deux chaînes ont au moins un caractère différent, False sinon.
- chaine1 `<` chaine2 : Renvoie True si la valeur de la chaîne chaine1 précède la valeur de la chaîne chaine2 dans l'ordre lexicographique (l'ordre du dictionnaire).
- chaine1 `<=` chaine2 : Comme ¡ mais la valeur des deux chaînes peuvent être les mêmes.
- chaine1 `>` chaine2 : Renvoie True si la valeur de la chaîne chaine1 suit la valeur de la chaîne chaine2 dans l'ordre lexicographique (l'ordre du dictionnaire).
- chaine1 `>=` chaine2 : Comme ¡ mais la valeur des deux chaînes peuvent être les mêmes.

Pour ranger dans l'ordre lexicographique, on compare les deux premiers caractères de chaque texte. S'ils sont égaux, on compare le deuxième, etc. Par exemple : "azerty" `<` "azfa" car les premiers termes de chaque chaîne sont égaux. De même pour le deuxième. Pour le troisième, comme "e" `<` "f", il en résultat que "azerty" `<=` "azfa". Il n'est pas nécessaire de regarder la suite.

## 5.1 Listes

Une liste est une structure de données qui contient une collection d'objets Python. Il s'agit d'un nouveau type par rapport aux entiers, float, booléens et chaînes de caractères que nous avons vus jusqu'à maintenant. On parle aussi d'objet séquentiel en ce sens qu'il contient une séquence d'autres objets.

Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. En voici quelques exemples :

Code Python

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ["girafe", 5, "souris", 0.15]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> tailles
[5, 2.5, 1.75, 0.15]
>>> mixte
['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

### 5.1.1 Utilisation

Un des gros avantages d'une liste est que vous accédez à ses éléments par leur position. Ce numéro est appelé **indice** (ou index) de la liste.

Code Python

```
liste  : ["girafe", "tigre", "singe", "souris"]
indice :        0        1        2         3
```

Soyez très attentif au fait que les indices d'une liste de n éléments commencent à ***0*** et se terminent à ***n-1***. Voyez l'exemple suivant :

Code Python

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

Code Python

```
>>> animaux[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

N'oubliez pas ceci ou vous risquez d'obtenir des bugs inattendus !

### 5.1.2 Opérations sur les listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication :

Code Python

```
>>> ani1 = ["girafe", "tigre"]
>>> ani2 = ["singe", "souris"]
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'opérateur + est très pratique pour concaténer deux listes.

Vous pouvez aussi utiliser la méthode **.append()** lorsque vous souhaitez ajouter un seul élément à la fin d'une liste.

Dans l'exemple suivant, nous allons créer une liste vide :

Code Python

```
>>> liste1 = []
>>> liste1
[]
```

puis lui ajouter deux éléments, l'un après l'autre, d'abord avec la concaténation :

Code Python

```
>>> liste1 = liste1 + [15]
>>> liste1
[15]
>>> liste1 = liste1 + [-5]
>>> liste1
[15, -5]
```

puis avec la méthode **.append()** :

Code Python

```
>>> liste1.append(13)
>>> liste1
[15, -5, 13]
>>> liste1.append(-3)
>>> liste1
[15, -5, 13, -3]
```

Dans cet exemple, nous ajoutons des éléments à une liste en utilisant l'opérateur de concaténation + ou la méthode **.append()**.

### 5.1.3 Indiçage négatif

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

Code Python

```
liste          : ["girafe", "tigre", "singe", "souris"]
indice positif :         0         1         2         3
indice négatif :        -4        -3        -2        -1
```

ou encore :

Code Python

```
liste          : ["A", "B", "C", "D", "E", "F"]
indice positif :   0    1    2    3    4    5
indice négatif :  -6   -5   -4   -3   -2   -1
```

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice -2, l'avant-avant dernier l'indice -3, etc. :

Code Python

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[-1]
'souris'
>>> animaux[-2]
'singe'
```

Pour accéder au premier élément de la liste avec un indice négatif, il faut par contre connaître le bon indice :

Code Python

```
>>> animaux[-4]
'girafe'
```

### 5.1.4 Tranches

Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant un indiçage construit sur le modèle [m:n+1] pour récupérer tous les éléments, du émième au énième (de l'élément m inclu

à l'élément n+1 exclu). On dit alors qu'on récupère une tranche de la liste, par exemple :

Code Python

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points **:**, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier :

Code Python

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

Finalement, on se rend compte que l'accès au contenu d'une liste

fonctionne sur le modèle ***liste[début:fin:pas]***.

### 5.1.5 Fonction *len()*

L'instruction ***len()*** vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

Code Python

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> len(animaux)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

### 5.1.6 Fonction *range()* et *list()*

L'instruction ***range()*** est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle. Lorsqu'elle est utilisée en combinaison avec la fonction **list()**, on obtient une liste d'entiers. Par exemple :

Code Python

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

### 5.1.7 Listes de listes

Pour finir, sachez qu'il est tout à fait possible de construire des listes de listes. Cette fonctionnalité peut parfois être très pratique. Par exemple :

Code Python

```
>>> prairie1 = ["girafe", 4]
>>> prairie2 = ["tigre", 2]
>>> prairie3 = ["singe", 5]
>>> savane = [prairie1, prairie2, prairie3]
>>> savane
[['girafe', 4], ['tigre', 2], ['singe', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie.

Pour accéder à un élément de la liste, on utilise l'indiçage habituel :

Code Python

```
>>> savane[1]
['tigre', 2]
```

Pour accéder à un élément de la sous-liste, on utilise un double indiçage :

Code Python

```
>>> savane[1][0]
'tigre'
>>> savane[1][1]
2
```

## 5.2 Les dictionnaires

Les dictionnaires sont des collections d'objets non-ordonnées. Un dictionnaire est composé d'éléments et chaque élément se compose d'une paire **clé: valeur**.

Dans d'autres langages de programmation, on parle de tableaux associatifs ou de hashs. Comme les listes, les dictionnaires sont des objets muables et dynamiques. Ils peuvent être modifiés et s'étendre selon vos besoins. Un dictionnaire peut contenir des objets de n'importe quel type et même inclure d'autres dictionnaires.

C'est grâce à ces caractéristiques que les dictionnaires sont souvent

utilisés pour créer des structures de données complexes où plusieurs éléments sont imbriqués les uns dans les autres !

### 5.2.1 Créer un dictionnaire

La façon la plus simple de créer un dictionnaire est d'ouvrir des accolades et d'y insérer des paires de clés et de valeurs.

Pour écrire une paire, il faut respecter la syntaxe suivante : **clé: valeur**.

Chaque paire doit être séparée de l'autre par une virgule. Les valeurs peuvent être de n'importe quel type alors que les clés doivent obligatoirement être de type immuable !

Vous pouvez ainsi utiliser un float ou un tuple comme clé sans problème :

Code Python

```
"""
# En théorie
d = {
        clé: valeur,
        clé: valeur,
        clé: valeur,
        ...
        clé: valeur
}
"""
# Dictionnaire vide

d = {}

print(d)

# Dictionnaire dont les clés ne sont que des chaînes de caractères
d = {
        'spam': 'eggs',
        'knights': 'lumberjack',
        'bacon': 'sausage'
}

print(d)

# Dictionnaire dont les clés sont des objets de différents types
d = {
        1: 'one',
        'deux': 2 ,
        (3, 4, 5): 'pas_de_soucis',
        9.9: 'nine_point_nine'
}
print(d)
```

Les dictionnaires qui viennent d'être créés n'ont aucun sens, c'est juste pour une application.

Vous pouvez aussi créer un dictionnaire grâce la fonction dict :

Code Python

```
d = dict() # {}

d = dict({
    'spam': 'eggs',
    'knights': 'lumberjack',
    'bacon': 'sausage'
})
print(d)
```

Une clé doit être unique, les doublons ne sont pas autorisés.

Si cela arrive, cela ne va pas créer d'erreur mais votre seconde clé écrasera la première.

Code Python

```
d = {
        'spam': 'eggs',
        'knights': 'lumberjack', # 1ère clé 'knights'
        'bacon': 'sausage',
        'knights': 'ham' # 2ème clé 'knights' qui va écraser la première
}

print(d)
```

Une clé ne peut pas être un objet muable :

Code Python

```python
d = {
        'spam': 'eggs',
        'knights': 'lumberjack',
        'bacon': 'sausage'
}

print(d['spam'])  # 'eggs'

# Si on essaie d'accéder à une clé inexistante avec les crochets, on a
↪  une erreur (KeyError)
# print(d['ham'])

print(d.get('spam'))  # 'eggs'

# Avec get, si la clé n'existe pas, on récupère None
print(d.get('ham'))  # None

# Ou la valeur par défaut que l'on passe en deuxième argument
print(d.get('ham', "Cette clé n'existe pas..."))  # "Cette clé n'existe
↪  pas..."
```

### 5.2.2 Accéder à un élément dans un dictionnaire

Alors qu'on accède aux éléments contenus dans une liste ou un tuple grâce à leurs indices (car ce sont des structures ordonnés), pour les dictionnaires, on utilise une clé.

On peut utiliser cette clé à l'intérieur de crochets [] ou via la méthode **get**. La différence entre les deux ? Avec les crochets, une erreur de type KeyError est levée si vous tentez d'utiliser une clé inexistante. Alors que la méthode **get** vous retournera simplement None ou un objet de votre choix.

Code Python

```
d = {
        'spam': 'eggs',
        'knights': 'lumberjack',
        'bacon': 'sausage'
}

print(d['spam'])  # 'eggs'

# Si on essaie d'accéder à une clé inexistante avec les crochets, on a
↪  une erreur (KeyError)
# print(d['ham'])

print(d.get('spam'))  # 'eggs'

# Avec get, si la clé n'existe pas, on récupère None
print(d.get('ham'))  # None

# Ou la valeur par défaut que l'on passe en deuxième argument
print(d.get('ham', "Cette clé n'existe pas..."))
# "Cette clé n'existe pas..."
```

### 5.2.3 Ajouter et modifier des éléments

Les dictionnaires étant des objets muables, ils sont faciles à modifier :

Code Python

```
d = {
    'spam': 'eggs',
    'knights': 'lumberjack',
}

d['spam'] = 'ham' # Clé existe déjà, remplace la valeur
d['bacon'] = 'sausage' # Nouvelle clé, on créer la paire clé/valeur dans
↪  le dico

print(d)
```

Cela signifie que vous pouvez aussi créer des dictionnaires à la volée

et y ajouter/supprimer des éléments en fonction de ce qu'il se passe dans votre code.

### 5.2.4 Supprimer des items d'un dictionnaire

Pour les suppressions, plusieurs possibilités en fonction de votre besoin :

Utiliser la méthode **pop** pour supprimer un élément et récupérer sa valeur dans une variable :

Code Python

```python
d = {
    'spam': 'ham',
    'knights': 'lumberjack',
    'bacon': 'sausage'
}

item = d.pop('knights') # 'lumberjack'
print(d)
print(item)
```

Utiliser la méthode **popitem** pour supprimer le dernier élément et récupérer un tuple contenant la clé et sa valeur :

Code Python

```python
d = {
    'spam': 'ham',
    'knights': 'lumberjack',
    'bacon': 'sausage'
}

item = d.popitem() # ('bacon', 'sausage') OU ('spam', 'ham') OU
↪  ('knights': 'lumberjack')
```

Utiliser **clear** pour vider le dictionnaire :

Code Python

```
d = {
    'spam': 'ham',
    'knights': 'lumberjack',
    'bacon': 'sausage'
}

d.clear() # {}
print(d)
```

Supprimer entièrement le dictionnaire grâce à l'instruction del :

Code Python

```
d = {
    'spam': 'ham',
    'knights': 'lumberjack',
    'bacon': 'sausage'
}

del d
print(d) # NameError: name 'd' is not defined
```

# Chapter 6

# La Régression Linéaire

La régression linéaire est une technique statistique utilisée pour modéliser et analyser la relation entre une variable dépendante (ou réponse) et une ou plusieurs variables indépendantes (ou explicatives). Elle permet de comprendre et de quantifier l'influence des variables indépendantes sur la variable dépendante.

## 6.1 Installer les bibliothèques nécessaires

Avant de commencer, assurez-vous d'avoir installé les bibliothèques nécessaires. Si ce n'est pas déjà fait, vous les installer en utilisant **pip** dans la console de Spyder :

Code Python

```
pip install numpy pandas scikit-learn matplotlib
```

## 6.2 Importer les bibliothèques

Nous allons commencer par importer les bibliothèques nécessaires pour la régression linéaire.

Code Python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

## 6.3 Charger les données

Nous allons utiliser le jeu de données Prestige.csv disponible en téléchargement sur Internet ici

Code Python

```
data = pd.read_csv("Prestige.csv")
```

ou encore

Code Python

```
data = pd.read_csv("Prestige.csv")
```

si le fichier Prestige.csv est dans votre répertoire de travail.

## 6.4 Visualiser les données

Avant de faire la régression, il est souvent utile de visualiser les données.

Code Python

```
plt.scatter(data['prestige'], data['education'])
plt.xlabel('Education')
plt.ylabel('Prestige')
plt.title('Visualisation des données')
plt.show()
```

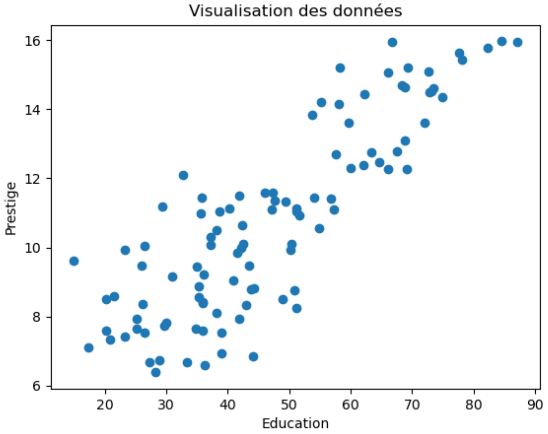


Figure 6.1: Visualisation de la linéarité entre les données

## 6.5 Préparer les données pour la modélisation

Nous devons diviser les données en ensembles d'entraînement et de test.

Code Python

```
X_train, X_test, y_train, y_test = train_test_split(
data[['education']], data['prestige'],test_size=0.2,
random_state=42)
```

## 6.6 Créer et entraîner le modèle de régression linéaire

Nous allons maintenant créer le modèle de régression linéaire et l'entraîner sur les données d'entraînement.

Code Python

```
# Créer le modèle de régression linéaire
model = LinearRegression()

# Entraîner le modèle
model.fit(X_train, y_train)
```

## 6.7 Faire des prédictions

Une fois le modèle de régression linéaire créé, vous pouvez l'utiliser pour faire des prévisions. Ceci est utile pour évaluer l'efficacité de votre modèle de régression linéaire en comparant les données prédites aux données normales du jeu de donnée.

Code Python

```
# Prédire les valeurs de y pour X_test
y_pred = model.predict(X_test)
```

## 6.8 Évaluer le modèle

Nous allons évaluer les performances du modèle en utilisant la métrique de l'erreur quadratique moyenne (MSE) et le coefficient de détermination ($R^2$).

Code Python

```
# Calculer l'erreur quadratique moyenne (MSE)
mse = mean_squared_error(y_test, y_pred)

# Calculer le coefficient de détermination (R^2)
r2 = r2_score(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
print(f'R^2 Score: {r2}')
```

Code Python

```
Mean Squared Error: 117.69641355665618
R^2 Score: 0.4712308384423046
```

Ces résultats pas du tout fameux nous permettent de comprendre que les deux jeux de données ne sont pas autant corrélées. On s'attend normalement à un $R^2$ d'au moins 0.75 pour des résultats acceptables. Ceci veut-dire que la variable ***Education*** n'explique pas assez la variable ***Prestige***.

## 6.9 Visualiser les résultats

Il est souvent utile de visualiser la ligne de régression par rapport aux données réelles.

Code Python

```
plt.scatter(X_test, y_test, color='black', label='Data')
plt.plot(X_test, y_pred, color='blue', linewidth=3,
label='Regression Line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Regression Line vs Data')
plt.legend()
plt.show()
```

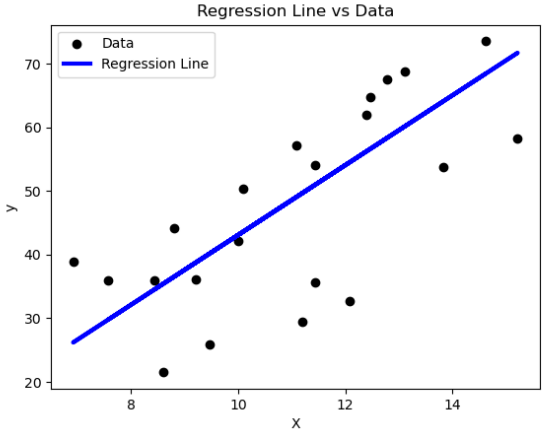


Figure 6.2: Modèle avec droite de régression linéaire

# Chapter 7

# La Régression Linéaire Multiple

Une régression linéaire permet d'expliquer, de manière linéaire, une variable à expliquer Y, aléatoire en fonction d'une ou plusieurs variables explicatives X. Il existe deux types de régression linéaire; simple qui contient une seule variable explicative, et multiple qui peut contenir plusieurs exogènes à la fois. Notre objectif dans ce cas-là, est de faire une prédiction des nouvelles valeurs en utilisant une régression linéaire multiple.

## 7.1 Description du jeu de données

L'ozone est un polluant photochimique, dont de nombreux instituts cherchent à prévoir les pics pour prévenir les populations. La pollution automobile, l'absence de vent et la chaleur comptent parmi les facteurs qui accroissent le taux de pollution.

*Téléchargez le jeu de données de l'ozone à cette adresse*
On considère à cet effet un jeu de données issu du Laboratoire de mathématiques appliquées de l'Agrocampus Ouest qui contient 112 données recueillies à Rennes durant l'été 2001. On y trouve les 14 variables suivantes :

- *obs* : mois-jour ;
- *maxO3* : teneur maximale en ozone observée sur la journée (en /gr/m3) ;

- *T9, T12, T15* : température observée à 9 h, 12 h et 15 h ;
- *Ne9, 12, Ne15* : nébulosité observée à 9 h, 12 h et 15 h ;
- *Vx9, Vx12, Vx15* : composante est-ouest du vent à 9 h, 12 h et 15 h ;
- *maxO3v* : teneur maximale en ozone observée la veille ;
- *vent* : orientation du vent à 12 h ;
- *pluie* : occurrence ou non de précipitations.

### 7.1.1 Objectifs de cette analyse

On souhaite étudier le lien entre le pic d'ozone journalier et un certain nombre de facteurs potentiellement explicatifs afin de proposer un modèle de régression permettant de prévenir la population.

Par exemple, la simple visualisation du nuage de points entre le pic d'ozone journalier et la température mesurée à 12 h permet d'ores et déjà d'envisager une relation linéaire :

$$maxO3 = \beta_1 + \beta_2 T12$$

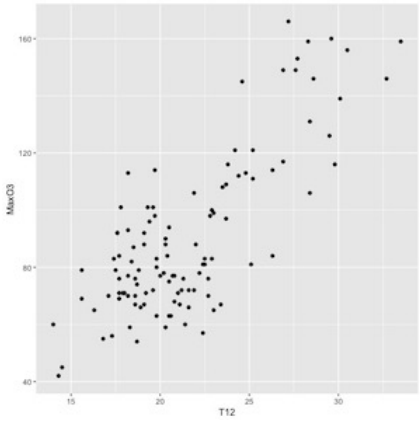


Figure 7.1: Ozone Nuage de Points

### 7.1.2 Importez les données

On importe les librairies qui nous permettront de mener à bien ce TP :

Code Python

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
from scipy.stats import t, shapiro
from statsmodels.stats.outliers_influence import
↪  variance_inflation_factor
import statsmodels
from functions import *
```

On importe les données, puis on utilise la commande lm pour régresser maxO3 en fonction des autres variables de l'échantillon.

Code Python

```
ozone = pd.read_csv('ozone.txt', sep=";", decimal=',')

reg_multi = smf.ols('maxO3~T9+T12+T15+Ne9+Ne12+Ne15+maxO3v',
↪  data=ozone).fit()
print(reg_multi.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  maxO3   R-squared:                       0.755
Model:                            OLS   Adj. R-squared:                  0.738
Method:                 Least Squares   F-statistic:                     45.68
Date:                Thu, 11 Jul 2024   Prob (F-statistic):           6.06e-29
Time:                        12:01:20   Log-Likelihood:                -453.71
No. Observations:                 112   AIC:                             923.4
Df Residuals:                     104   BIC:                             945.2
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
```

| | | | | | | |
|---|---|---|---|---|---|---|
| Intercept | 12.7055 | 13.109 | 0.969 | 0.335 | -13.289 | 38.700 |
| T9 | -0.6360 | 1.035 | -0.615 | 0.540 | -2.688 | 1.416 |
| T12 | 2.5060 | 1.399 | 1.791 | 0.076 | -0.269 | 5.281 |
| T15 | 0.7138 | 1.137 | 0.628 | 0.531 | -1.540 | 2.968 |
| Ne9 | -2.7606 | 0.892 | -3.096 | 0.003 | -4.529 | -0.993 |
| Ne12 | -0.3719 | 1.346 | -0.276 | 0.783 | -3.041 | 2.297 |
| Ne15 | 0.0903 | 0.999 | 0.090 | 0.928 | -1.891 | 2.072 |
| maxO3v | 0.3777 | 0.061 | 6.171 | 0.000 | 0.256 | 0.499 |

| | | | |
|---|---|---|---|
| Omnibus: | 10.038 | Durbin-Watson: | 1.895 |
| Prob(Omnibus): | 0.007 | Jarque-Bera (JB): | 23.403 |
| Skew: | -0.139 | Prob(JB): | 8.28e-06 |
| Kurtosis: | 5.222 | Cond. No. | 979. |

On constate ici que certains paramètres ne sont pas significativement différents de 0, car leur p-valeur (P>|t|) n'est pas inférieure à 0.05, le niveau de test que nous souhaitons.

Le $R^2$ vaut environ 0.75 et le $R^2$ ajustévaut environ 0.74.

Cette valeur est significativement plus élevée que ce qu'on a eu en Régression Linéaire Simple quoique les jeux de données soient différents. Mais c'est logique, car lorsque l'on rajoute des variables explicatives potentielles, on accroît naturellement la valeur de $R^2$.

### 7.1.3 Retirez les variables non significatives

Il se trouve qu'au rang des 7 variables explicatives utilisés pour la modélisation, toutes les variables ne contribuent pas vraiment à l'explication de la variable 'maxO3'.Ceci est principalement dû à une faible correlation de ces variables avec la variable 'max03'. Il est mieux de retirer ces variables puisqu'elle ne servent pas à grand-chose.
On va donc maintenant retirer les variables non significatives. On commence par la moins significative : Ne15, car elle a la p-value la plus élevée (0.92).

Code Python

```
reg_multi = smf.ols('maxO3~T9+T12+T15+Ne9+Ne12+maxO3v',
↪  data=ozone).fit()
print(reg_multi.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   maxO3   R-squared:                       0.755
Model:                             OLS   Adj. R-squared:                  0.741
Method:                  Least Squares   F-statistic:                     53.80
Date:                 Tue, 12 Mar 2019   Prob (F-statistic):           7.91e-30
Time:                         14:02:23   Log-Likelihood:                -453.71
No. Observations:                  112   AIC:                             921.4
Df Residuals:                      105   BIC:                             940.5
Df Model:                            6
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     12.8492     12.950      0.992      0.323     -12.829      38.527
T9            -0.6298      1.027     -0.613      0.541      -2.667       1.407
T12            2.5602      1.258      2.034      0.044       0.065       5.055
T15            0.6579      0.949      0.693      0.490      -1.223       2.539
Ne9           -2.7653      0.886     -3.122      0.002      -4.522      -1.009
Ne12          -0.3080      1.139     -0.270      0.787      -2.567       1.951
maxO3v         0.3775      0.061      6.202      0.000       0.257       0.498
==============================================================================
Omnibus:                        9.980   Durbin-Watson:                   1.894
Prob(Omnibus):                  0.007   Jarque-Bera (JB):               23.200
Skew:                          -0.136   Prob(JB):                     9.17e-06
Kurtosis:                       5.213   Cond. No.                         971.
==============================================================================
```

On voit alors que c'est maintenant Ne12, avec une p-valeur de 0.79, qui est la moins significative. On l'enlève donc.

Code Python

```
reg_multi = smf.ols('maxO3~T9+T12+T15+Ne9+maxO3v', data=ozone).fit()
print(reg_multi.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   maxO3   R-squared:                       0.754
Model:                             OLS   Adj. R-squared:                  0.743
Method:                  Least Squares   F-statistic:                     65.11
```

```
Date:                Tue, 12 Mar 2019   Prob (F-statistic):           9.62e-31
Time:                        14:02:23   Log-Likelihood:                -453.75
No. Observations:                 112   AIC:                             919.5
Df Residuals:                     106   BIC:                             935.8
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     11.2844     11.534      0.978      0.330     -11.583      34.152
T9            -0.7313      0.952     -0.768      0.444      -2.619       1.157
T12            2.6649      1.192      2.235      0.027       0.301       5.028
T15            0.6682      0.944      0.708      0.481      -1.203       2.539
Ne9           -2.9258      0.655     -4.470      0.000      -4.223      -1.628
maxO3v         0.3796      0.060      6.314      0.000       0.260       0.499
==============================================================================
Omnibus:                        9.521   Durbin-Watson:                   1.896
Prob(Omnibus):                  0.009   Jarque-Bera (JB):               21.507
Skew:                          -0.116   Prob(JB):                     2.14e-05
Kurtosis:                       5.134   Cond. No.                         868.
==============================================================================
```

On constate qu'il faut maintenant retirer la variable T9 :

Code Python

```
reg_multi = smf.ols('maxO3~T12+T15+Ne9+maxO3v', data=ozone).fit()
print(reg_multi.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  maxO3   R-squared:                       0.753
Model:                            OLS   Adj. R-squared:                  0.744
Method:                 Least Squares   F-statistic:                     81.55
Date:                Tue, 12 Mar 2019   Prob (F-statistic):           1.33e-31
Time:                        14:02:23   Log-Likelihood:                -454.06
No. Observations:                 112   AIC:                             918.1
Df Residuals:                     107   BIC:                             931.7
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
```

```
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      9.1368     11.168      0.818      0.415     -13.003      31.277
T12            2.2318      1.048      2.129      0.036       0.154       4.310
T15            0.6277      0.941      0.667      0.506      -1.237       2.492
Ne9           -2.9639      0.651     -4.550      0.000      -4.255      -1.673
maxO3v         0.3702      0.059      6.301      0.000       0.254       0.487
==============================================================================
Omnibus:                        8.655   Durbin-Watson:                   1.844
Prob(Omnibus):                  0.013   Jarque-Bera (JB):               18.693
Skew:                          -0.035   Prob(JB):                     8.73e-05
Kurtosis:                       5.000   Cond. No.                         828.
==============================================================================
```

Et l'on retire ensuite T15 :

Code Python

```
reg_multi = smf.ols('maxO3~T12+Ne9+maxO3v', data=ozone).fit()
print(reg_multi.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  maxO3   R-squared:                       0.752
Model:                            OLS   Adj. R-squared:                  0.745
Method:                 Least Squares   F-statistic:                     109.1
Date:                Tue, 12 Mar 2019   Prob (F-statistic):           1.46e-32
Time:                        14:02:23   Log-Likelihood:                -454.30
No. Observations:                 112   AIC:                             916.6
Df Residuals:                     108   BIC:                             927.5
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      9.7622     11.100      0.879      0.381     -12.241      31.765
T12            2.8531      0.481      5.937      0.000       1.901       3.806
Ne9           -3.0242      0.643     -4.700      0.000      -4.300      -1.749
maxO3v         0.3757      0.058      6.477      0.000       0.261       0.491
==============================================================================
Omnibus:                        9.766   Durbin-Watson:                   1.884
```

```
Prob(Omnibus):                  0.008   Jarque-Bera (JB):                23.605
Skew:                           0.020   Prob(JB):                     7.48e-06
Kurtosis:                       5.249   Cond. No.                         805.
==============================================================================
```

On remarque qu'à présent, tous les paramètres sont significatifs. Quant au $R^2$, il vaut environ 0.75, tout comme le $R^2$ ajusté.

On peut donc utiliser ce modèle à des fins de prévision !

Si l'on souhaite prévoir la concentration journalière en ozone, sachant que la température prévue à 12 h sera de 15 °C, que la valeur de Ne9 sera de 2, et que la concentration maxO3v de la veille vaut 100, alors on saisit les lignes suivantes :

Code Python

```
a_prevoir = pd.DataFrame({'T12': 15, 'Ne9': 2, 'maxO3v': 100},
↪  index=[0])
maxO3_prev = reg_multi.predict(a_prevoir)
print(round(maxO3_prev[0], 2))

84.08
```

On obtient une concentration maxO3 de 84

### 7.1.4 Testez la normalité des données

Si l'on veut tester la normalité des résidus, on peut faire un test de Shapiro-Wilk.

Code Python

```
shapiro(reg_multi.resid)

(0.9623235464096069, 0.0030248425900936127)
```

Ici, l'hypothèse de normalité est remise en cause (p-value = 0.003 ¡ 0.05).

Néanmoins, l'observation des résidus, le fait qu'ils ne soient pas très différents d'une distribution symétrique, et le fait que l'échantillon soit de taille suffisante (supérieure à 30) permettent de dire que les résultats

obtenus par le modèle linéaire gaussien ne sont pas absurdes, même si le résidu n'est pas considéré comme étant gaussien.

### 7.1.5 Testez l'homoscédasticité

On peut également tester l'homoscédasticité (c'est-à-dire la constance de la variance) des résidus :

Code Python

```
_, pval, __, f_pval =
↪  statsmodels.stats.diagnostic.het_breuschpagan(reg_multi.resid,
↪  variables)
print('p value test Breusch Pagan:', pval)

p value test Breusch Pagan: 0.07865197865995636
```

La p-valeur ici n'est pas inférieure à 0.05, on ne rejette pas l'hypothèse selon laquelle les variances sont constantes (l'hypothèse d'homoscédasticité).

### 7.1.6 Vérifier la colinéarité des variables

Une autre chose à vérifier est l'éventuelle colinéarité approchée des variables :

Code Python

```
variables = reg_multi.model.exog
[variance_inflation_factor(variables, i) for i in
↪  np.arange(1,variables.shape[1])]

[2.0678328061249003, 1.5277727396873015, 1.4747178841142814]
```

Ici, tous les coefficients sont inférieurs à 10, il n'y a donc pas de problème de colinéarité.