# Algorithmique et programmation

Yann Thorimbert

Année scolaire 2024-2025

# Préface

Ce document a été rédigé pour servir de support de cours aux élèves d'informatique de première année (et en partie seconde année) du collège de Genève. Il incorpore les éléments du plan d'études relatifs à la programmation et, dans une moindre mesure, à l'algorithmique.

Bien qu'il se veuille indépendant de tout autre support, ce cours va de pair avec des documents et explications orales fournies par l'enseignant. En particulier, des exercices de consolidation, explications orales supplémentaires et autres retours personnalisés sont nécessaires pour que la matière du cours soit complètement acquise par un élève.

La branche informatique étant nouvellement introduite au moment de la rédaction de ce document (2021), le lecteur est invité à signaler toute erreur ou question afin de participer à l'amélioration du polycopié : `yann.thorimbert@gmail.com`.

# Table des matières


I Programmation 6

1 Introduction 7

1.1 Qu'est-ce que la programmation ? 7

1.2 Les langages de programmation 8

1.2.1 Qu'est-ce qu'un langage de programmation ? 8

1.2.2 Le langage Python 8

1.3 Avantages et désavantages de Python 10

1.4 Exécuter un code et afficher un message 11

1.5 Développer un code 11

1.5.1 Les commentaires 11

1.5.2 L'ordre d'exécution des instructions 12

2 Les variables 13

2.1 Déclarer des variables 13

2.1.1 Comment nommer les variables ? 15

2.1.2 Nom, valeur et assignation 16

2.2 Les types de variables 17

2.2.1 Les nombres entiers 17

2.2.2 Les nombres à virgule 18

2.2.3 Les textes 18

2.2.4 Les booléens 19

2.2.5 L'utilisation des f-strings . . . 19

2.2.6 Afficher le type de variable . . . 19

2.3 Les conversions de type . . . 21

2.4 Les opérations sur les variables . . . 22

2.5 Les opérateurs d'assignement . . . 24

**3 Les conditions** **26**

3.1 Les opérateurs de comparaison . . . 27

3.2 Les opérateurs logiques . . . 27

3.3 Utilisation du `elif` . . . 28

3.4 Conditions imbriquées . . . 31

**4 Les boucles** **34**

4.1 Les boucles `while` . . . 35

4.1.1 Structure de base . . . 35

4.1.2 Les boucles « infinies » . . . 35

4.2 Les boucles `for` . . . 36

4.2.1 Structure de base . . . 36

**5 Les listes** **40**

5.1 Déclarer une liste et accéder à ses éléments . . . 40

5.2 Parcourir une liste . . . 41

5.3 Manipuler une liste . . . 43

5.4 Itération sur les éléments d'une liste . . . 45

5.5 Accéder à des sous-listes . . . 45

**6 Les fonctions** **47**

6.1 Les arguments d'une fonction . . . 48

6.2 Valeur de retour . . . 50

**7 Manipulation des chaînes de caractères** **52**

**8 Nombres pseudo-aléatoires** **57**

8.1 Génération d'un `int` aléatoire . . . . . . . . . . . . . . . . . . . . . . . . . 57

8.2 Génération d'un `float` aléatoire . . . . . . . . . . . . . . . . . . . . . . . . 59

**9 Pour aller plus loin** **61**

9.1 Les dictionnaires . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

9.2 Les classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

9.2.1 Définition d'une classe . . . . . . . . . . . . . . . . . . . . . . . . 65

9.3 Les modules et les packages . . . . . . . . . . . . . . . . . . . . . . . . . 66

9.4 Gestion des environnements avec venv . . . . . . . . . . . . . . . . . . . 68

9.5 Les graphiques avec Matplotlib . . . . . . . . . . . . . . . . . . . . . . . 68

9.6 Écriture et lecture de fichiers . . . . . . . . . . . . . . . . . . . . . . . . . 71

9.6.1 Lecture et écriture de fichiers texte . . . . . . . . . . . . . . . . . 71

9.6.2 Traitement des exceptions . . . . . . . . . . . . . . . . . . . . . . 72

9.6.3 Écriture et lecture de structures de données avec Pickle . . . . . . . 72

**II Algorithmique** **75**

**10 Introduction à l'algorithmique** **76**

10.1 Qu'est-ce qu'un algorithme ? . . . . . . . . . . . . . . . . . . . . . . . . . 76

10.2 Un premier exemple . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

10.3 Exercices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**11 Stratégies gloutonnes** **80**

11.1 Exemple de la traversée du désert . . . . . . . . . . . . . . . . . . . . . . 80

11.2 Exercices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

11.3 Tri de liste . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

11.3.1 Tri par sélection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
11.3.2 Tri à bulles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

**12 Diviser pour régner** **87**

12.1 Principe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
12.1.1 Recherche dichotomique . . . . . . . . . . . . . . . . . . . . . . . . 87
12.1.2 Comparaison avec un algorithme glouton . . . . . . . . . . . . . . . 89
12.1.3 Complexité et performances . . . . . . . . . . . . . . . . . . . . . . 90
12.2 Tri fusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

# Première partie

# Programmation

# Chapitre 1

# Introduction

## 1.1 Qu'est-ce que la programmation ?

Comme nous l'avons vu lors des premiers cours de l'année, « informatique » est un terme très large qui désigne, dans la langue française, une multitude de domaines différents :

- La bureautique (traitement de textes, diapositives, tableurs, ...)
- La technologie des ordinateurs ou « hardware », c'est-à-dire ce qui constitue les ordinateurs au niveau matériel.
- La science informatique et l'algorithmique, c'est-à-dire les mathématiques qui servent à décrire un ordinateur et les instructions qu'il peut exécuter.
- **La programmation, c'est-à-dire l'art de programmer des logiciels [1].**

Dans cette partie du cours, c'est ce dernier point qui va nous intéresser. Nous allons apprendre à donner des instructions à un ordinateur afin qu'il exécute certaines tâches automatiquement ou en réaction à nos agissements. Par exemple : écrire une application qui indique la bonne quantité d'ingrédients pour une recette ; écrire un mini jeu-vidéo ; écrire une application qui permet d'afficher automatiquement à l'écran le barème des points d'une épreuve.

En programmation et dans ce cours, nous allons souvent utiliser deux termes : programmeur et utilisateur. Le **programmeur** décide des instructions à donner à l'ordinateur afin de construire une application. L'**utilisateur** est la personne qui utilise l'application. L'utilisateur ne voit pas les instructions que le programmeur a écrites. L'utilisateur n'a pas besoin de connaître un langage de programmation ; le programmeur, oui !

Exemple : les programmeurs de l'application « Horloge » sur votre téléphone ont écrit un code dans un langage de programmation. Lorsque vous utilisez l'application pour régler votre réveil-matin [2], vous ne voyez pas le code des programmeurs. Vous utilisez simplement l'application sans devoir réfléchir a ce qui est « caché » derrière.

1. Rappel : dans ce cours, « logiciel » et « application » sont des synonymes !
2. Car vous ne souhaitez pas arriver en retard à votre cours d'informatique, par exemple.

## 1.2 Les langages de programmation

Il existe plein de façons différentes de faire de la programmation. En effet, il existe non seulement une multitude de langages de programmation, mais en plus il existe de nombreux styles de programmation. Dans ce cours, nous allons adopter un langage et un style de programmation donnés. Il faut garder à l'esprit que ce n'est là qu'un aperçu du monde de la programmation.

### 1.2.1 Qu'est-ce qu'un langage de programmation ?

Un langage de programmation définit une façon de donner des instructions à l'ordinateur. Nous avons vu que l'ordinateur travaillait avec des nombres en binaire. Comment passer d'un ordre en français, comme par exemple « colorier en bleu le pixel en haut à gauche de l'écran », à un ordre en binaire que l'ordinateur peut comprendre ? C'est en grande partie le langage de programmation qui nous permet de le faire.

Tout comme il existe plusieurs façons de dire le mot « pomme » à travers le monde, il existe plusieurs façons d'ordonner à l'ordinateur de colorier en bleu le pixel en haut à gauche de l'écran.

Voici un premier exemple d'instruction que l'on peut donner à l'ordinateur. Dans le bout de code ci-dessous, on demande à la machine d'afficher "Bonjour tout le monde" dans le **terminal**.

```
print("Bonjour tout le monde")
```

Cette instruction a été donnée à l'ordinateur dans le langage Python. Il est en mesure de l'exécuter s'il comprend ce langage.

### 1.2.2 Le langage Python

Python est un langage de programmation parmi d'autres. Ses deux caractéristiques principales sont :

- Il est très répandu de nos jours dans le monde professionnel et le monde académique.

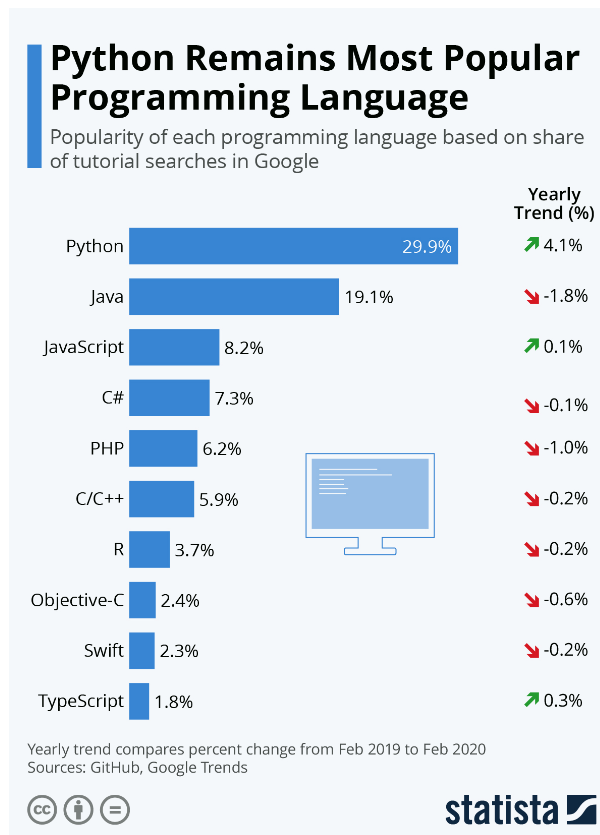



FIGURE 1.1 – Popularité de différents langages mesurée comme le nombre de recherches de tutoriaux sur le web, en 2020. Source : statista.com

- C'est un langage dit de « haut niveau ». Cela veut dire que, lorsqu'on programme en Python, on n'a en général moins besoin de se soucier de ce qu'il se passe exactement dans le processeur et dans la mémoire vive que lorsqu'on programme avec un langage « bas niveau » comme le C ou le C++. En Python, on est en quelque sorte plus proche du langage des humains que de celui de la machine. Pour certaines tâches, Python est donc plus facile d'utilisation.

**Pour aller plus loin : les couches d'abstraction**

En Fig. 1.2 sont montrées les couches de langages pour passer d'un ordre en langage naturel (par exemple le français) à un ordre exprimé en binaire pour la machine. Les instructions qui sont exprimées dans une couche doivent êtres traduites dans la couche du dessous pour que la chaîne fonctionne.

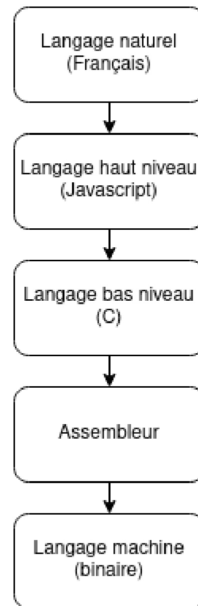



FIGURE 1.2 – Couches de langages de programmation pour passer du langage naturel au langage de la machine, avec des exemples entre parenthèses. Python est un langage de haut niveau, comme Javascript par exemple.

## 1.3 Avantages et désavantages de Python

Au moment où ce document est écrit (2024), Python est largement le langage le plus recherché sur Google[3].

**Avantages de Python :**

- Python est installé par défaut sur la plupart des OS modernes.
- C'est un langage qui permet de facilement écrire des programmes **multi plateformes** : il faut donc peu se soucier de savoir si l'utilisateur de votre code utilise Windows, MacOS, Ubuntu, iOS, Android, etc.
- Il existe une grande **communauté** de codeurs en Python, ce qui signifie que l'aide et la documentation sont faciles à trouver sur le web.
- Python impose une façon de coder très stricte, qui permet dans certains contextes de rendre le code plus **lisible** que dans d'autres langages de programmation.

**Désavantages de Python :**

- Comme la plupart des langages haut niveau, Python peut s'avérer peu performant pour certaines tâches (cela nous importe peu dans ce cours).
- Comme le langage est moins rigoureux que d'autres, il est possible de contracter de mauvaises habitudes de programmation en Python plus que dans d'autres langages, notamment en ce qui concerne le type des données (voir chapitre 2).

3. `http://pypl.github.io/PYPL.html`

## 1.4 Exécuter un code et afficher un message

**Exercice 1.0**

Écrivez votre premier code.

1. Créez un fichier nommé `bonjour.py` dans votre espace personnel.
2. Ouvrez le fichier avec un éditeur de texte et écrivez :

```
print("Hello, world")
```

3. Avec un terminal (suivre les instructions de l'enseignant) exécutez le script que vous venez d'écrire.

L'exercice précédent permet d'obtenir un message dans votre terminal. Ce message contient le texte que vous avez indiqué dans votre code Python (aussi nommé « script »).

**Exercice 1.1**

Éditez le script de l'exercice précédent et changez le texte du message. Ajoutez d'autres messages de type « print » et vérifiez qu'ils apparaissent dans l'ordre.

## 1.5 Développer un code

### 1.5.1 Les commentaires

Il est impératif de laisser des notes claires sur la logique du code que vous écrivez. Cela aide les autres à comprendre votre code (on code rarement seul dans la réalité). Cela vous aide également à comprendre votre propre code lorsque celui-ci est très grand et complexe, ou que vous n'avez pas travaillé dessus depuis quelques temps.

Les commentaires servent aux programmeurs qui lisent le code. Au moment d'exécuter les instructions, l'ordinateur ignore toujours les commentaires ; vous pouvez donc les écrire comme bon vous semble pourvu qu'ils soient clairs et utiles.

#### Commentaire sur une ligne

Pour insérer un commentaire simple, on utilise le dièse « # » :

```
print("Bonjour...") #Ceci est mon premier commentaire !
print("Au-revoir !") #Ceci est mon second commentaire...
#print("Rebonjour !") #Cette instruction a été commentée.
```

### Commentaire multilignes

Pour insérer un commentaire sur plusieurs lignes, on peut entre autres utiliser les triple guillemets `"""` ou ''' avant et après la portion de code à ignorer :

```
print("Hello, world.")
"""Bonjour,
Bienvenu dans mon code.
Ceci est un commentaire sur plusieurs lignes.
Les lignes de commentaires sont ignorées au moment de l'exécution.
On peut s'en servir pour expliquer le code."""
print("Au-revoir !")
```

En plus d'expliquer le code, les commentaires sont souvent utilisés par les programmeurs pour garder « de côté » des bouts de code sans avoir à les effacer. Il suffit de commenter ou « décommenter » les lignes de code en question pour obtenir ou enlever un effet, lorsqu'on débogue notre application.

La plupart des éditeurs de code (comme Thonny, par exemple) proposent des raccourcis clavier pour commenter/décommenter d'un seul coup plusieurs lignes. Il est capital de connaître ces raccourcis pour ne pas perdre de temps à commenter les lignes unes à unes !

## 1.5.2 L'ordre d'exécution des instructions

Regardons le code suivant, où le numéro des lignes a été indiqué à gauche pour vous aider (mais ces numéros ne font pas partie du code).

```
print("Ceci est le premier message.")
print("Ceci est le second message.")
print("Ceci est le troisième message.")
```

Si l'on exécute ce code, on observe que les instructions sont exécutées les unes après les autres, de la première à la dernière ligne en commençant par le haut. On dit alors que les instructions sont exécutés **séquentiellement**. De nos jours, de nombreux codes sont également écrits pour être exécutés en **parallèle** (c'est-à-dire non-séquentiellement). Dans ce cours, nous considérerons toujours des codes séquentiels.

# Chapitre 2

# Les variables

Le but de la programmation impérative [1] étant de pouvoir décrire des « recettes » à l'ordinateur, il faut pouvoir parler des différents objets sur lesquels porte la recette. C'est le rôle des variables. Dans ce chapitre, nous allons voir comment déclarer des variables, comment les manipuler et comment les utiliser dans des opérations mathématiques.

On peut s'imaginer les variables comme des boîtes contenant des nombres ou des objets qui nous intéressent. Ces boîtes ont un nom et contiennent une valeur.

Dans de nombreux langages adaptés aux débutants, les variables sont cachées au programmeur. Par exemple, dans les langages de type « tortue » [2] que nous avons déjà expérimentés, on utilise rarement des variables : il suffit de dire à la tortue où elle doit aller, quand elle doit tourner, etc.

Supposons maintenant que nous écrivions une applications très simple. L'utilisateur de l'application est invité à insérer son nom, puis un message est diffusé à l'écran. Si l'utilisateur s'appelle Alice, alors le message affiché doit être : « Bonjour, Alice ! ».

Le code de cette application contiendra au moins deux lignes :

1. Demander son nom à l'utilisateur, et le **retenir** en mémoire.
2. **Afficher** le texte à l'écran, en utilisant le nom qui a été retenu.

Lorsque nous demandons à l'ordinateur d'afficher le prénom précédemment renseigné, il a fallut que ce dernier soit retenu quelque part dans la mémoire de l'ordinateur. C'est pourquoi nous avons utilisé une variable.

## 2.1 Déclarer des variables

Pour utiliser une variable, il faut d'abord la **déclarer**.

1. Nous ne traitons pas dans ce cours de certains paradigmes de programmation tels que la programmation fonctionnelle, pour lesquels le concept de variable revêt une importance différente de celle que nous considérons ici.
2. Voir `https://turtleacademy.com/` ou `https://www.thorpy.org/progid` par exemple.

Une déclaration de variable ressemble toujours à ceci :

```
nom_de_la_variable = valeur_de_la_variable
```

### Exercice 2.0

Écrivez un script contenant le code suivant :

```python
nom = "Alice"
age = 16
print("L'utilisateur se nomme ", nom, " et il a ", age, " ans.")
```

Observez le résultat et modifiez le code pour qu'il affiche dans la console « L'utilisateur se nomme Bob et il a 17 ans ».

Dans l'exercice précédent, nous avons déclaré deux variables. La première contient le nom de l'utilisateur tandis que la seconde contient son âge. On commence toujours par donner le nom de la variable (par exemple `age`). Ensuite, on indique la **valeur** de la variable après le signe égal.

### Pour aller plus loin : plusieurs instructions par ligne

Il est possible, bien que déconseillé dans ce cours, de donner plusieurs instructions par ligne de code. Pour cela, il faut utiliser le point-virgule, comme ceci :

```python
a = 15; print(a); b = 12; print(b)
```

Dans beaucoup d'autres langages de programmation, le point-virgule est obligatoire pour signaler la fin d'une instruction. Cependant, en Python, un retour à la ligne indique implicitement une fin d'instruction.

### Exercice 2.1

Modifiez le code de l'exercice précédent afin qu'il affiche le nom de deux personnes ainsi que leur âge. Stockez les deux noms et âges dans des variables différentes (vous devez déclarer 4 variables en tout).

### Exercice 2.2

Écrivez un script qui demande à l'utilisateur de rentrer son nom, puis affiche un message de type `print` disant : « Bonjour, ... » avec le nom de la personne en lieu et place des points de suspension. Partez du code suivant :

```python
nom = input("Quel est votre nom ?")
```

### Pour aller plus loin : d'autres façons de déclarer des variables

Il est également possibles de déclarer plusieurs variables à la fois :

```
nom, age, surnom = "Robert", 16, "Bob"
```

Dans ce cours, nous déclarerons en général une seule variable à la fois.

### 2.1.1 Comment nommer les variables ?

#### Règles de nommage

Voici les règles à connaître pour que le nom des variables soit valide en Python (et dans la plupart des langages modernes) :

- Le nom d'une variable ne doit jamais commencer par un chiffre. Dans ce cours, le nom des variables commençera *toujours* par une lettre.
- Le nom d'une variable ne doit jamais être l'un des mots-clé du langage. Par exemple, on ne peut pas appeller une variable « if ».
- Le nom d'une variable peut contenir des lettres ou des chiffres (sauf si c'est la première lettre). Il peut également contenir les caractères underscore (« _ ») ainsi que d'autres caractères que nous n'utiliserons pas dans ce cours.
- La casse compte (par exemple, les variables `Age` et `age` ont des noms différents).

#### Conventions de nommage et bonnes pratiques

En plus des règles mentionnées ci-dessus, nous allons nous astreindre à certaines bonnes pratiques :

- Le nom d'une variable doit être **clair et parlant**. Par exemple, dans le code du dernier exercice, « `nom` » est clair et parlant, tandis que « `n` » ne l'est pas.
- En respectant le point précédent, le nom d'une variable doit être aussi **bref** que possible.
- Le nom des variables commence par une minuscule. Dans ce cours, si le nom de la variable est un mot composé, on séparera chacun de ses termes par un underscore. Par exemple : `nom_utilisateur = "Alice"`.

**Exercice 2.3 - Exercice non programmatique**

Parmi les déclarations suivantes, lesquelles sont valides ? Répondez dans votre tête. Pour vérifier votre réponse, vous pouvez recopier le code et voir s'il génère une erreur.

```
age utilisateur = 12
print(age utilisateur)
```

```
age_utilisateur = 12
print(age_utilisateur)
```

```
perimetre = 4
deuxFoisPerimetre = 8
```

```
perimetre = 4
2FoisPerimetre = 8
```

```
perimetre = 4
perimetreFois2 = 8
```

```
perimetre = 4
print(Perimetre)
```

### 2.1.2 Nom, valeur et assignation

Dans tous les exemples précédents, le signe « `=` » a été utilisé pour signifier à l'ordinateur une instruction d'**assignation**. L'assignation est le fait de donner une valeur à une variable. Par exemple, dans le code suivant, on déclare en première ligne la variable `a` à laquelle on assigne la valeur 3. En seconde ligne, on lui assigne une nouvelle valeur, égale à 7.

```
a = 3
a = 7
```

Il faut faire très attention ici : **le signe d'égalité n'a pas la même signification qu'en mathématique**. En effet, en Python ce qui se trouve à gauche du signe d'égalité est toujours une variable, tandis que ce qui se trouve à droite est toujours la nouvelle valeur à assigner à cette variable. Ainsi, on peut écrire `a = a + 3` pour ajouter la valeur 3 à ce qui se trouve déjà au sein de la variable `a`. Cette dernière égalité n'est pas vraie dans un sens mathématique ; ce n'est pas une égalité à vrai dire, c'est une assignation ! Pour cette raison, certains langages ont même adopté un autre symbole que le signe d'égalité pour l'assignation, comme la flèche. Dans ces langages, on écrirait `a` ← `a + 3` pour ajouter

3 à `a`.

**Pour aller plus loin : les constantes**

Parfois, il est très pratique de signifier que la valeur d'une variable ne doit pas changer. Dans les codes longs ou compliqués, cela peut beaucoup aider le programmeur. Par exemple, le nombre $\pi \approx 3.141$ est une constante mathématique ; sa valeur ne change pas d'un moment à l'autre. C'est donc une bonne idée de le définir comme telle dans le code :

```
PI = 3.141
```

Si l'on essaie de modifier la valeur de `PI` après la première ligne dans le code ci-dessus, on générera une erreur. Nous verrons dans les prochains chapitres des types de variables dont le contenu peut changer même si elles sont constantes. Il faudra donc être attentif au type de variable que l'on choisit de déclarer comme constantes.

## 2.2 Les types de variables

Dans les exemples de la section précédente, certaines variables contenaient le nom d'une personne (par exemple « Alice »), tandis que d'autres contenaient leur âge (par exemple « 12 »).

Un langage de programmation doit distinguer différents **types** de données. Dans le code ci-dessous, on peut distinguer quatre types de variable qui sont tout le temps utilisés en programmation :

```
a = 2.718 #ceci est un nombre à virgule (float)
b = "Alice" #ceci est une chaîne de caractères (str)
c = True #ceci est un booléen (bool)
d = 15 #ceci est un nombre entier (int)
```

Lorsque vous programmez, **vous devez toujours être capable de dire quel est le type de chaque variable que vous manipulez**. Ceci est capital pour toute la suite du cours.

### 2.2.1 Les nombres entiers

En Python, leur nom est simplement `int`, qui est l'abréviation de « integer » en anglais. Voici un exemple contenant des déclarations de nombres entiers :

```
a = 2
b = 3
c = 1000
d = -12
```

### 2.2.2 Les nombres à virgule

Dans de nombreux langages importants, il existe une différence entre les nombres entiers et les nombres à virgule (« float » en anglais). Ces derniers servent à approximer les nombres réels.

Voici un exemple contenant des déclarations de nombres à virgule :

```
a = 2.4
b = 3.0
c = 1000.2555
d = -12.623
```

Par ailleurs, on peut déclarer un nombre sous la forme de notation scientifique de la façon suivante :

```
n = 3.2e8
```

Cela signifie : $n = 3,2 \cdot 10^8$. Il est également possible de représenter l'infini de la façon suivante :

```
mon_nombre_infini = float("inf")
```

### 2.2.3 Les textes

Pour représenter un texte, on utilise des **chaînes de caractères** (abrégé « string » en anglais). Cela désigne simplement une suite de caractères (lettres, chiffres et tout autre symbole défini dans une convention comme Unicode). Une chaîne de caractères est délimitée par des **guillemets** ou des **apostrophes**. Voici un exemple contenant des déclarations de textes :

```
a = "Alice"
b = 'Bob'
c = "Bonjour à tous !"
d = "Mon nombre préféré est le 3"
```

Ainsi, les deux lignes suivantes ne désignent pas la même chose :

```
a = 2.718
b = "2.718"
```

La variable `a` contient un nombre, tandis que la variable `b` contient un texte (même s'il se trouve que ce texte désigne un nombre) ! Le chapitre 7 aborde les chaînes de caractères plus en détail.

### 2.2.4 Les booléens

Un variable booléenne ne peut prendre que deux valeurs : « True » ou « False ». C'est donc une variable binaire. Sur le Web, il est courant de trouver des personnes qui utilisent le nombre 1 pour signifier `True` et le nombre 0 pour signifier `False`. Nous verrons dans le chapitre sur les **conditions** que les booléens sont extrêmement utilisés en programmation !

> **Pour aller plus loin : la taille d'un booléen**
>
> Comme un booléen ne peut prendre que deux valeurs possibles, il est théoriquement encodable sur un seul bit. Pour des raisons pratiques, nos ordinateurs encodent tout de même les booléens sur au moins un octet ; selon l'ordinateur et la version de Python, cela peut même prendre huit octets ! On fait ce « gaspillage » pour des raisons complexes d'architecture et de performances, et parce que les ordinateurs modernes ont une mémoire qui nous le permet.

### 2.2.5 L'utilisation des f-strings

Depuis la version 3.6 de Python, il est possible d'utiliser les f-strings pour afficher des variables dans des chaînes de caractères. Pour comprendre l'un des intérêts de cette méthode, considérons le code suivant, qui utilise la fonction `print` pour afficher le contenu de trois variables :

```
nom = "Alice"
age = 16
taille = 165
print("L'utilisateur", nom, "a", age, "ans. Il mesure", taille, "cm.")
```

Voici une version équivalente utilisant les f-strings :

```
nom = "Alice"
age = 16
taille = 165
print(f"L'utilisateur {nom} a {age} ans. Il mesure {taille} cm.")
```

Les f-strings sont de plus en plus souvent utilisés au sein de la communauté, aussi est-il important de les connaître ne serait-ce que pour comprendre le code des autres (notamment sur les forums ou les outils d'IA capables de générer du code). Ils permettent bien d'autres options de formatage des données, que nous n'aborderons pas dans ce cours.

### 2.2.6 Afficher le type de variable

On peut accéder au type d'une variable à tout moment grâce à la fonction `type` de Python, comme dans l'exemple ci-dessous.

```python
a = 2.718 #ceci est un nombre à virgule (float)
b = "Alice" #ceci est une chaîne de caractères (string)
c = True #ceci est un booléen
d = 3 #ceci est un nombre entier (int)
print(type(a)) #on affiche le type de la variable a.
print(type(b)) #on affiche le type de la variable b.
print(type(c)) #on affiche le type de la variable c.
print(type(d)) #on affiche le type de la variable d.
```

En cas de doute, n'hésitez pas à afficher un message dans la console pour vous aider à déboguer votre application. Lorsque vous programmez, vous devez toujours connaître le type des variables que vous manipulez – dans le cas contraire, cela signifie que vous ne comprenez potentiellement pas le sens des transformations que vous leur appliquez !

**Pour aller plus loin : d'autres types d'objets**

Nous verrons plus loin comment définir des objets de type liste ou de type dictionnaire en Python. Par ailleurs, il est courant en programmation de définir de nouveaux types de données. En particulier, un courant de programmation particulièrement populaire dans les dernières décennies et nommé la « programmation orientée objet » se base beaucoup sur la création de nouveaux types de variables par le programmeur. Dans ce cours, nous ne traiterons pas de la programmation orientée objet.

**Exercice 2.4 - Exercice non programmatique**

Le code ci-dessous déclare plusieurs variables. Pour chacune de ces variables, écrivez son type en commentaire. Pour tester votre réponse, il vous suffit de copier le code dans un fichier Python et de demander à afficher le type de chaque variable.

```python
a = 2
b = "Bob"
c = False
d = 2.222
e = 0.0001
f = 10000
g = "   "
h = "Salut tout le monde."
i = "True"
j = 0
k = 1
```

## 2.3 Les conversions de type

Il est possible de convertir une variable d'un type en une variable d'un autre type. En anglais, on parle de « type casting ». Cela est très utile dans certains cas, par exemple lorsqu'on veut recueillir des informations de l'utilisateur :

```
n_utilisateur = input("Entrez un nombre") #équivaut à str(input("..."))
n_plus_trois = int(n_utilisateur) + 3
print("Votre nombre plus trois vaut", n_plus_trois)
```

La ligne 2 du code précédent générerait une erreur si la variable entrée par l'utilisateur n'était pas convertie en nombre entier grâce à l'instruction `int(...)`. En effet, on essayerait alors d'ajouter un nombre à une chaîne de caractère, ce qui n'a pas de sens ! Parfois, cependant, un oubli de conversion peut causer des problèmes plus subtils, comme dans l'exercice suivant.

**Exercice 2.5**

Exécutez le code suivant et trouvez un moyen de le déboguer.

```
n_utilisateur = input("Entrez un nombre")
n_fois_deux = n_utilisateur * 2
print("Le double de votre nombre vaut", n_fois_deux)
```

Voici un tableau qui récapitule, pour chacun des types discutés jusqu'ici, l'opérateur qui lui est associé.

| Type de variable | Opérateur Python |
|---|---|
| Chaîne de caractères | `str` |
| Nombre entier | `int` |
| Nombre à virgule | `float` |
| Booléean | `bool` |

Par exemple, pour convertir le nombre 45.28 en une chaîne de caractères, il suffit d'écrire `str(45.28)`, tandis que `int(45.28)` a pour effet d'arrondir 45.28 à l'entier inférieur. Toutes les conversions ne fonctionnent pas, cependant. Par exemple, `int("Bonjour")` génère une erreur.

## 2.4 Les opérations sur les variables

Il est naturel de vouloir effectuer des opérations sur les variables. Par exemple, écrivons un code qui convertit les minutes en secondes :

```
nb_minutes = 3
nb_sec = nb_minutes * 60
print(nb_minutes, " min = ", nb_sec , " sec.")
```

**Exercice 2.6**

1. Reprenez le code ci-dessus pour afficher le nombre de secondes correspondant au nombre de minutes introduit par l'utilisateur avec la fonction `input` utilisée plus haut.
2. Demandez à l'utilisateur d'introduire un nombre d'heures. Affichez à l'écran un message pour l'utilisateur lui donnant le nombre de secondes correspondant.

Comme on vient de le voir dans l'exemple ci-dessus, l'opération « * » indique une multiplication. Cet opérateur est défini sur les nombres : cela signifie que les variables de type `number` peuvent être multipliées entre elles. En revanche, la multiplication de deux variables de type `string` n'est pas définie ! Attention cependant, car la multiplication entre un `int` et un `string` est définie en Python : par exemple `3 * "hi"` est égal à `"hihihi"`.

Voici un tableau qui résume les opérateurs mathématiques que nous allons utiliser dans ce cours :

| Symbole | Nom de l'opérateur |
|---|---|
| `+` | addition |
| `-` | soustraction |
| `*` | multiplication |
| `/` | division |
| `**` | puissance |
| `%` | modulo |

Dans le tableau ci-dessus, nous n'avons pas indiqué les booléens car ils sont traités comme les nombres 0 et 1 lorsqu'on utilise des opérateurs mathématiques sur eux.

À noter que les seuls opérateurs mathématiques du tableau qui soient défini pour les chaînes de caractères sont les additions, qui ont pour effet de concaténer deux chaînes

entre elles, et la multiplication, qui a pour effet de répéter une chaîne un nombre entier de fois, comme discuté plus haut.

**Pour aller plus loin : l'opérateur modulo**

L'opérateur modulo représente le **reste** de la division entière. Par exemple : `12 % 4 = 0` car le reste de la division de 12 par 4 vaut zéro. De même `11 % 3 = 2` car le reste de la division de 11 par 3 vaut 2.
L'opérateur modulo est très utilisé en programmation. Il permet par exemple de tester si un nombre quelconque $x$ est multiple de 2. En effet, si $x$ % 2 vaut zéro, cela signifie que $x$ est divisble par 2 (autrement dit : c'est un nombre pair).

**Exercice 2.7**

Écrivez un programme qui demande un nombre à l'utilisateur et affiche ensuite la valeur du cube de ce nombre à l'écran (message `print`).

**Exercice 2.8 - Théorème de Pythagore automatisé**

Écrivez une application qui demande à l'utilisateur d'entrer les catètes d'un triangle rectangle. Ensuite, la longueur de l'hypoténuse est affichée à l'écran. Pour calculer la racine carrée de `x`, on peut exécuter l'instruction `x**0.5`.

**Exercice 2.9 - Exercice non programmatique ( !)**

L'ordre des opérations suit les mêmes règles qu'en mathématiques. Cela signifie qu'il faut parfois utiliser des parenthèses pour indiquer à l'ordinateur le calcul que vous désirez lui faire faire. Considérons le code suivant :

```
a = 12
b = 2
c = 3
d = 5
# Quelle est la valeur des variables ci-dessous ?
e = a*b + d
f = a*b - c*d
g = b*(c + d)*2
h = a/b + a/c
i = h**b
```

Quel est la valeur des variables `e` à `i` ? Écrivez la réponse en commentaire des variables, puis vérifiez vos réponses en affichant la valeur des variables.

**Exercice 2.10 - Formule de Viète automatisée**

Écrivez une application qui demande à l'utilisateur d'entrer les valeurs $a$, $b$ et $c$ d'une équation du second degré du type $ax^2 + bx + c = 0$. Le programme affiche ensuite la valeur de $\Delta = b^2 - 4ac$ ainsi que les solutions s'il y en a. Les solutions sont : $x_{\pm} = \frac{-b \pm \sqrt{\Delta}}{2a}$. Comme précédemment, voici un exemple de racine carrée : le code `x**0.5` permet de calculer $\sqrt{x}$.

**Pour aller plus loin : les opérations du module Math**

La plupart des opérations mathématiques usuelles sont définies dans un **module** externe. Un module est un code déjà écrit par d'autres gens et que l'on peut utiliser dans notre programme. Voici quelques exemples d'utilisation du module `Math` :

```
import math #il faut toujours importer le module pour commencer
alpha = 0.1 # angle en radians
sin_alpha = math.sin(alpha) # calcule le sinus de alpha
cos_2pi = math.cos(2*math.pi) #calcule le cosinus de 2 pi
```

Nous utiliserons très peu le module Math dans ce cours car beaucoup des fonctions mathématiques qu'il définit ne sont étudiées au collège qu'en deuxième année ou plus tard, dans le cours de mathématiques. Dans tous les cas, il ne faut pas hésiter à consulter la documentation des modules sur le web.

## 2.5 Les opérateurs d'assignement

Une fois que nous aurons vu le chapitre sur les conditions, il sera pratique de pouvoir **incrémenter** une variable. Prenons l'exemple suivant :

```
a = 3
#Imaginons qu'à la place de ce commentaire l'utilisateur fasse quelque
#chose. En réaction à cela, nous voulons ajouter 7 à la valeur de a.
a = a + 7 #On ajoute 7 à la valeur de a.
```

Dans le code ci-dessus, l'instruction `a = a + 7` pourrait être remplacée par `a += 7`. Il est très courant d'exprimer l'incrémentation de cette manière. En particulier, sur l'Internet, vous trouverez beaucoup ce type d'instruction. Voici des exemples supplémentaires incluant d'autres opérateurs d'assignement :

```
1  a = 18
2
3  a *= 2 #Cela a le même effet que a = a*2.
4  #La variable a vaut maintenant 36.
5
6  a /= 3 #Cela a le même effet que a = a/3.
7  #La variable a vaut maintenant 12.
```

```
a += 3 #Cela a le même effet que a = a + 3.
#La variable a vaut maintenant 15.

a -= 5 #Cela a le même effet que a = a - 5.
#La variable a vaut maintenant 10.
```

# Chapitre 3

# Les conditions

L'un des ingrédients qui nous manquent pour pouvoir écrire n'importe quel programme est la capacité de notre code à réagir à son environnement. Supposons qu'Alice indique son âge à l'ordinateur ; comment faire pour afficher un message donné si Alice est mineure, mais un autre message si elle est majeure ?

Une condition s'écrit avec le mot-clé **if**. C'est le premier mot-clé que nous voyons dans ce cours. Un mot-clé ne peut être utilisé comme nom de variable car il a déjà une signification particulière au sein du langage. La syntaxe d'utilisation de `if` est la suivante (ici, nous utilisons des `print` pour montrer où doit se trouver le code de l'utilisateur) :

```
if condition:
    print("Bonjour")#instruction à exécuter si la condition est vérifiée.
```

Une seconde syntaxe possible est :

```
if condition:
    print("Bonjour")#instruction à exécuter si la condition est vérifiée.
else:
    print("Ciao")#à exécuter si la condition n'est pas vérifée.
```

Dans le dernier exemple, un autre mot-clé du langage a été utilisé : **else**. Dans tous les exemples ci-dessus on voit que des instructions ont été **indentées**. Les indentations servent à délimiter le code qui est concerné par la condition. Voici un exemple complet que vous pouvez tester :

```
age = int(input("Quel est l'âge d'Alice ?"))
if age < 18:
    print("Alice est mineure.")
else:
    print("Alice est majeure.")
```

Si l'indentation des instructions n'est pas respectée, on génère des erreurs d'indentation, voire pire, des incohérences logiques dans le code.

**Pour aller plus loin : le retour à la ligne est-il obligatoire ?**

Dans certains cas, il est possible (mais pas recommandé) d'indiquer une instruction sur la même ligne qu'une condition. Par exemple, dans le code suivant, le premier message est soumis à condition, tandis que le second message sera toujours affiché :

```
n = int(input("Insérez un nombre"))
if n > 10: print("Vous avez inséré un nombre plus grand que 10")
print("Fin du programme.")
```

## 3.1 Les opérateurs de comparaison

Dans l'exemple avec Alice au début de ce chapitre, nous avons comparé l'âge d'Alice à un nombre afin de réagir en conséquence. Nous avons utilisé le symbole « < ». Il existe d'autres opérateurs de comparaison :

| Symbole | Signification |
|---|---|
| == | Égal à |
| != | Différent de |
| < | Plus petit que |
| <= | Plus petit ou égal à |
| > | Plus grand que |
| >= | Plus grand ou égal à |

**Exercice 3.0 - Exercice non programmatique**

Quel est l'output du code ci-dessous ? Écrivez en commentaire le résultat attendu, puis testez le code pour vérifier vos réponses.

```
if 12 < 3:
    print("A")
else:
    print("B")
if 12/2 == 6:
    print("C")
```

## 3.2 Les opérateurs logiques

Supposons maintenant que nous voulions écrire un message à Alice si son âge est supérieur à 18 ans et que sa taille est supérieure à 150 cm. Ce seraient, par exemple, les

conditions à remplir pour monter dans un manège particulièrement dangereux d'un parc d'attractions. Un code possible est :

```
age = int(input("Quel est l'âge d'Alice ?"))
taille = int(input("Quelle est la taille (en cm) d'Alice ?"))
if age >= 18 and taille > 150:
    print("Alice peut monter dans le manège.")
else:
    print("Désolé, Alice ne peut pas monter dans le manège.")
```

Il existe de nombreux opérateurs logiques, mais nous n'en utiliserons que deux dans ce cours : le ET logique et le OU logique.

**Exercice 3.1**

Écrivez un programme qui demande un nombre à l'utilisateur. Si le nombre entré par l'utilisateur est plus grand que 10 et plus petit que 20, le message suivant doit être affiché : « Votre nombre est situé entre dix et vingt ».

**Exercice 3.2 - Exercice non programmatique**

Prévoyez quel sera l'output du code ci-dessous. Ensuite, testez le code pour vérifier vos réponses.

```
a = 12
b = 6
if a > b and b*2 == a :
    print("A")
else:
    print("B")
if 12/2 == b or b > 20:
    print("C")
else:
    print("D")
```

## 3.3 Utilisation du `elif`

Il est possible d'utiliser une troisième syntaxe pour exprimer les conditions :

```
if condition1:
    #instruction à exécuter si condition1 est vérifiée.
elif condition2:
    #instruction à exécuter si condition1 n'est pas vérifiée et que
    #condition2 est vérifiée
else:
```

```
    #instruction à exécuter dans les autres cas
```

Il est à noter qu'on peut enchaîner autant de conditions que l'on veut, bien que l'exemple ci-dessus n'en montre que deux. Attention : le code à l'intérieur du `else` n'est exécuté que dans le cas où aucune des conditions qui le précédent n'est vérifiée. Si l'une de ces conditions est vérifiée, le code dans le `else` ne sera pas exécuté.

Le `elif` est utile dans les cas où l'on énumère des possibilités. Par exemple, si l'on veut dicter un odre de préférence pour une liste de courses, on pourrait obtenir une phrase comme ceci : « Vas faire les courses. S'il y a du jus de pomme, achète-en. Sinon, s'il y a du jus d'orange, achète-en. Sinon, achète de l'eau. » Une telle phrase peut se résumer par deux structures de code différentes :

Première méthode (sans `elif`, mais en imbriquant des conditions) :

```
produit = input("Qu'y a-t-il de disponible parmi les 3 boissons ?")
if produit == "jus de pomme":
    print("On achète du jus de pomme")
else:
    if produit == "jus d'orange":
        print("On achète du jus d'orange")
    else:
        print("On achète de l'eau")
```

Seconde méthode (avec `elif`) :

```
produit = input("Qu'y a-t-il de disponible parmi les 3 boissons ?")
if produit == "jus de pomme":
    print("On achète du jus de pomme")
elif produit == "jus d'orange":
    print("On achète du jus d'orange")
else:
    print("On achète de l'eau")
```

En résumé :

1. Une suite de conditions commence toujours par un `if`.
2. Une suite de conditions peut se terminer par un `if`, un `elif` ou un `else`.
3. un `elif` ou un `else` se réfèrent toujours à la condition qui la précède au sein du même niveau hiérarchique.
4. Un `elif` ou un `else` n'est pris que dans le cas où aucune des conditions du même niveau hiérarchique qui précède n'a été prise.

### Exercice 3.3

Écrivez un programme qui demande à l'utilisateur d'entrer un nombre. Le programme indique ensuite si le nombre est positif, négatif ou nul.

### Exercice 3.4 - Distributeur automatique

Dans un distributeur automatique, le chocolat coût 3 chf, l'eau coûte 1.5 chf et les chips coûtent 5 chf.
Écrivez un programme qui demande à l'utilisateur un nom de produit à acheter. Si le nom du produit est autre chose que « chocolat », « eau » ou « chips », le message « Erreur : nous n'avons pas ce produit » s'affiche. Sinon, le prix du produit s'affiche.
Attention : dans tous les cas, le programme doit se terminer par le message « Merci, aurevoir », même si le produit n'était pas disponible.

### Exercice 3.5 - Distributeur automatique avec code secret

Même exercice que le précédent, mais cette fois on demande également un code secret à l'utilisateur. Si le code est le bon (il vaut « xb37vk8 »), alors tous les produits sont à moitié prix.

### Exercice 3.6 - Exercice non programmatique

Quel est l'output du code ci-dessous ? Écrivez en commentaire le résultat attendu, puis testez le code pour vérifier vos réponses.

```
a = 1
b = 3
if 3*b == a:
    print("A")
elif 3*b == 8:
    print("B")
elif 2*b == 6:
    print("C")
elif a == 1:
    print("D")
print("Z")
if a == 1:
    print("E")
else:
    print("F")
print("G")
```

## 3.4 Conditions imbriquées

Il est possible d'imbriquer des conditions les unes dans les autres. Observons le code suivant :

```
age = 18
taille = 150
if age < 18:
    if taille > 150:
        print("Vous pouvez monter accompagné d'un adulte.")
    else:
        print("Vous ne pouvez pas monter dans le manège.")
else:
    print("Vous pouvez monter dans le manège.")
```

Ce code correspond à la situation où, si la personne est majeure, elle peut monter dans le manège ; si elle n'est pas majeure elle ne peut monter dans le manège que si sa taille est supérieure à 150 cm.

**Pour aller plus loin : les diagrammes de type « algorigramme »**

Pour représenter un **algorithme**, il est parfois plus pratique de le faire sous la forme d'un diagramme. Par exemple, l'effet du code précédent peut s'exprimer comme suit :

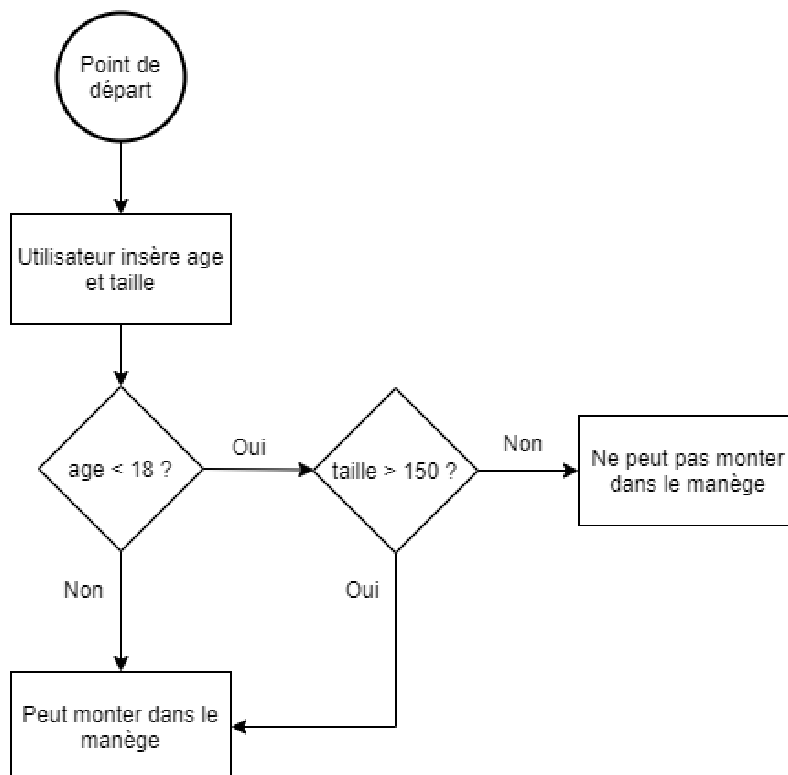



FIGURE 3.1 – Exemple de diagramme résumant un algorithme pour décider si une personne peut entrer dans un manège.

### Exercice 3.7 - Exercice non programmatique

Quel est l'output du code ci-dessous ? Écrivez en commentaire le résultat attendu, puis testez le code pour vérifier vos réponses.

```
a = 1
b = 3
if 3*a == b:
    print("A")
    if 2*b == 6:
        print("B")
        if b+2 == 4:
            print("C")
        else:
            print("D")
    print("E")
else:
    print("F")
print("G")
```

### Exercice 3.8

Écrivez un programme qui demande à l'utilisateur d'entrer le jour de la semaine (par exemple `"mardi"`) et une heure de la journée (par exemple `14`). Le programme indique ensuite à l'utilisateur s'il a congé, sachant que l'utilisateur a congé le mercredi à partir de 12h, tout le samedi et tout le dimanche. Les autres jours, il travaille entre 8h et 17h.

### Exercice 3.9

Écrivez un programme qui demande à l'utilisateur d'entrer trois nombres. Le programme doit ensuite afficher les trois nombres triés dans l'ordre croissant.

### Pour aller plus loin : utiliser les booléens

Les comparaisons utilisent le résultats d'opérateurs de comparaison tels que « est plus grand que » ou « est égal à ». La réponse à une question du type « 2 est-il plus grand que 3 ? » n'a que deux réponses possibles : oui ou non. Le résultat d'une comparaison est donc un booléen. Voici un exemple où l'utilisation des booléens permet de rendre le code plus lisible :

```
a = 3
b = 10
c = 7
condition1 = a*10 > b
condition2 = b/a > c
condition3 = c*c > b - 4*a
if condition1 and (condition2 or condition3):
    print("A")
#Ci-dessous, un code equivalent, moins lisible :
if a*10 > b and (b/a > c or c*c > b - 4*a):
    print("A")
```

Dans bien d'autres cas que nous ne montrons pas ici, les booléens peuvent être utiles.

# Chapitre 4

# Les boucles

Le dernier ingrédient qui nous manque pour écrire n'importe quel programme est la capacité à **répéter** des instructions automatiquement.

Prenons l'exemple suivant : l'utilisateur est invité à entrer un nombre. S'il entre le nombre à deviner (décidons que ce nombre vaut 66, par exemple), alors le programme est terminé. Tant que l'utilisateur n'entre pas le nombre à deviner, il peut continuer à entrer un nombre.

Écrire un tel programme en utilisant uniquement les outils que nous avons vus jusqu'à maintenant est impossible : on ne connaît pas à l'avance le nombre que l'utilisateur va entrer, on ne sait donc pas combien d'instructions il faut écrire (ce nombre est potentiellement infini). Un tel programme correspondrait au diagramme suivant :

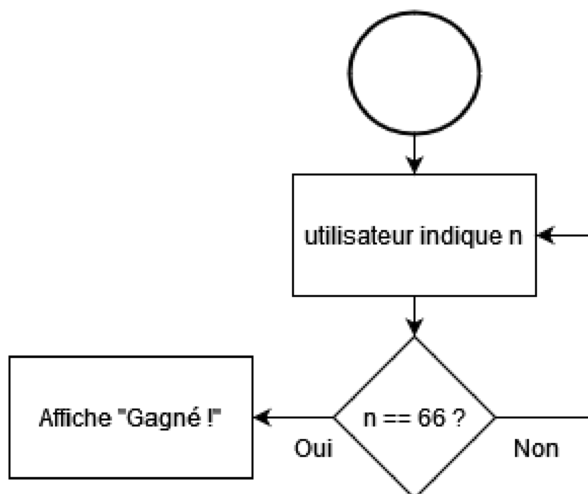



Figure 4.1 – Diagramme correspondant au jeu du nombre à deviner. La branche « non » du test `n==66` effectue une boucle vers un point précédent du programme.

La solution à ce problème réside dans l'utilisation des **boucles**. Nous allons voir deux types de boucles : les boucles **while** et les boucles **for**. Les boucles et les conditions (voir chapitre prédécent) constituent ce que l'on appelle des **structures de contrôle**.

## 4.1 Les boucles `while`

### 4.1.1 Structure de base

Le mot-clé `while` permet d'indiquer à l'ordinateur qu'il faut répéter certaines instructions **tant que** certaines conditions sont remplies. Voici une façon de résoudre, à l'aide d'une telle boucle, le problème donné en introduction de ce chapitre (jeu du nombre à deviner) :

```
n = int(input("Entrez un nombre."))
while n != 66:
    n = int(input("Essayez encore..."))
print("Gagné !")
```

Dans le code précédent, la condition de la boucle est que la variable `n` soit différente de 66. Tant que cette condition est vraie, le code à l'intérieur de la boucle est exécuté. La ligne 5 (message au vainqueur) ne peut donc être affichée que si `n` vaut 66.

**Exercice 4.0**

Créez un programme qui affiche tous les nombres de 1 à 100 en utilisant une boucle `while`. Dessinez sur une feuille l'algorigramme qui correspond à votre programme.

**Exercice 4.1**

Créez une application qui affiche tous les nombres de $a$ à $b$ en utilisant une boucle `while`. Les nombres $a$ et $b$ sont entrés par l'utilisateur via un `input` au début de l'application.

**Exercice 4.2**

Créez le programme suivant : on demande à l'utilisateur d'entrer un nombre, puis on affiche tous les multiples de trois jusqu'à ce nombre. Par exemple, si l'utilisateur entre « 37 », alors le programme doit afficher « 3 », puis « 6 », puis « 9 » et ainsi de suite jusqu'à « 36 ». Dessinez sur une feuille le diagramme algorigramme qui correspond à votre programme.

### 4.1.2 Les boucles « infinies »

Il est très courant de devoir écrire une boucle qui peut, potentiellement, être exécutée une infinité de fois, à l'image de la boucle du code précédent (jeu du deviner le nombre). Le menu d'un jeu vidéo, par exemple, est une boucle infinie : tant que l'utilisateur ne décide pas de quitter le menu, le jeu continue. En règle générale, il vaut mieux écrire directement dans la condition du `while` le test qui détermine si la boucle doit être poursuivie ou pas. Cependant, dans certaines conditions, on peut demander à quitter la boucle grâce

au mot-clé `break`. Voici un exemple qui implémente le jeu du nombre à deviner à l'aide d'une boucle infinie.

```
while True: #cette condition sera toujours vraie !
    n = int(input("Entrez un nombre."))
    if n == 66: #si l'utilisateur a entré 66...
        break #on quitte la boucle !
print("Gagné !")
```

**Exercice 4.3 — Difficile**

Créez une application de calculatrice qui permet de faire des multiplications. Tant que l'utilisateur n'entre pas le mot « stop », l'application reste ouverte et l'utilisateur peut entrer deux nombres dont on affiche le produit à l'écran.

## 4.2 Les boucles `for`

Le mot-clé **for** permet d'écrire un type de boucle différent des boucles `while`. En Python, les boucles `for` ont surtout été créées pour parcourir les listes, que nous verrons dans le prochain chapitre (ainsi que tout autre type de variable dites « itérable » que nous ne traitons pas dans ce cours). Cependant, afin de respecter la progression fixée par le plan d'études, nous traitons les boucles for avant de traiter des listes.

### 4.2.1 Structure de base

Une boucle `for` dépend de la valeur d'une **variable de boucle**. Voici un premier exemple de boucle `for`, où l'on choisit de nommer « i » la variable de boucle, et où l'on décide de répéter 10 fois une même instruction. Attention : `i` étant une variable, vous pouvez lui donner le nom que vous voulez et pas nécessairement « i ».

```
1 for i in range(10):
2     print("Tour de boucle numéro ", i)
```

Si l'on teste le code précédent, on s'aperçoit qu'il écrit dans la console « Tour de boucle numéro 0 », puis « Tour de boucle numéro 1 » et ainsi de suite jusqu'à « Tour de boucle numéro 9 ». Cela fait donc 10 répétitions d'une même instruction. La ligne `for i in range(10):` peut se comprendre comme « répéter les instructions tant que i est plus petit que 10, en commençant avec i=0, et en faisant `i=i+1` à chaque nouveau tour de boucle ».

**NB :** Les boucles `for` s'avèreront particulièrement utiles au moment de manipuler des objets tels que les listes, abordées dans le prochain chapitre.

### Exercice 4.4

Écrivez une boucle for qui affiche tous les nombres de 0 à 99.

### Exercice 4.5

Écrivez une boucle for qui affiche un nombre sur deux de 0 à 98, en commençant par 0.

### Exercice 4.6

Proposez un diagramme logigramme pour l'exercice précédent.

### Exercice 4.7

Écrivez une boucle for qui affiche tous les multiples de 6 jusqu'à 120.

### Exercice 4.8

Écrivez une boucle for qui affiche tous les multiples de $x$ jusqu'à 100, où $x$ est un nombre entré par l'utilisateur.

### Exercice 4.9

Créez une application qui affiche tous les nombres de $a$ à $b$ en utilisant une boucle `for`. Les nombres $a$ et $b$ sont entrés par l'utilisateur via un `input` au début de l'application.

### Exercice 4.10

À l'aide d'une boucle for, vérifiez si un nombre entier entre 0 et 10'000 est la solution de l'équation $3 \cdot x + 2 = 35$. Affichez ce nombre s'il est trouvé.

### Exercice 4.11

À l'aide d'une boucle for, vérifiez si un nombre entier entre -100'000 et 100'000 est la solution de l'équation $5 \cdot x + 269900 = 355$. Affichez ce nombre s'il est trouvé.

### Exercice 4.12 — Difficile (niveau M2)

À l'aide d'une boucle for, vérifiez cherchez le nombre entier entre -100'000 et 100'000 qui est le plus proche de résoudre l'équation $17 \cdot x + 1000 = 3$. Affichez ce nombre s'il est trouvé.

### Exercice 4.13 — Difficile (niveau M2)

À l'aide d'une boucle for, trouvez la somme de tous les nombres entiers entre 1 et $N$ compris, où $N$ est spécifié par l'utilisateur.

### Exercice 4.14 — Difficile (niveau M2)

À l'aide d'une boucle for, trouvez le produit de tous les nombres entiers entre 1 et $N$ compris, où $N$ est spécifié par l'utilisateur.

### Exercice 4.15 — Difficile (niveau M2)

À l'aide d'une boucle for ou while, trouvez le plus grand diviseur commun entre deux nombres spécifiés par l'utilisateur.

### Exercice 4.16 — Difficile (niveau M2)

À l'aide d'une boucle for ou while, affichez dans la console le barème complet pour une épreuve comptant $N$ points ($N$ est donné par l'utilisateur). La note $n$ à une épreuve est calculée comme suit :

$$n = \frac{x}{N} \cdot 5 + 1,$$

où $x$ est le nombre de points obtenus.

### Exercice 4.17 — Difficile (niveau M2)

Un paquet de pâtes coûte 4 CHF, un pack d'oeufs coût 3 CHF et une brique de lait coûte 3,50 CHF. Sachant que l'on est revenu des courses avec une facture de 32.5 CHF et que l'on a acheté uniquement les produits décrits ci-dessus, trouver toutes les possibilités d'achats auxquelles cette facture correspond. Ecrivez un programme utilisant plusieurs boucles imbriquées.

### Exercice 4.18 — Difficile (niveau M2)

Écrivez un code qui affiche un "carré" d'étoiles à l'écran. Le nombre d'étoiles sur un côté du carré est spécifié par l'utilisateur. Exemple de résultat avec un carré de 3 étoiles de côté :

```
***
***
***
```

### Exercice 4.19 — Difficile (niveau M2)

Écrivez un code qui affiche un "sapin" d'étoiles à l'écran. La largeur de la base du sapin est spécifiée par l'utilisateur et doit toujours être impaire (écrire un message d'erreur si ce n'est pas le cas). Exemple de résultat avec une base de 7 étoiles.

```
   *
  ***
 *****
*******
```

### Exercice 4.20 — Difficile (niveau M2)

À l'aide d'une boucle, construisez une seule chaîne de caractères qui contient toute la table de multiplication de 6 jusqu'à 60. Affichez ensuite la table avec un seul message de type print (la table entière doit s'afficher d'un seul coup à l'écran).

### Exercice 4.21 — Très difficile

À l'aide de deux boucles for imbriquées l'une dans l'autre, créez un programme qui affiche tous les nombres premiers jusqu'à 1000.

# Chapitre 5

# Les listes

Lorsque l'on veut créer une application où un traitement est effectué automatiquement et de façon répétitive sur un grand nombre de données, il est très pratique de pouvoir stocker ces données au sein d'une seule et même variable. En Python, une telle variable est une liste [1].

De fait, les listes sont un outil extrêmement utilisé lorsqu'on programme en Python. Dans beaucoup de cas, elles permettent de rendre le code plus court et plus simple à comprendre.

## 5.1 Déclarer une liste et accéder à ses éléments

Imaginons une application qui permette à l'utilisateur de noter une liste de ses prénoms préférés. Par exemple :

```
prenoms = ["Alice", "Albert", "Zoe", "Coralie"]
```

Les crochets sont utilisés pour indiquer à Python que les différentes variables (ici ce sont des strings, mais cela pourrait être n'importe quel autre type de variable !) sont stockées dans une liste.

Pour **accéder** aux éléments d'une liste, on utilise les crochets également. Par exemple, pour afficher le prénom préféré numéro 2, on écrit :

```
print(prenoms[2]) #cela affiche "Zoe" !
```

Attention, le tout premier élément d'une liste est **zéro**. La case numéro 2 désigne donc le 3ème élément. En informatique, il est très courant que les listes (ou tableaux) commencent avec l'indice numéro zéro. Une conséquence de cela est qu'au sein d'une liste contenant $N$ éléments, les éléments sont numérotés de 0 à $N-1$.

1. En réalité il existe bien d'autres structures de données utilisables selon le contexte, mais la liste est la plus simple d'entre elles.

**Exercice 5.0**

Créez une liste de strings contenant cinq éléments. Chaque élément doit être un nom de pays. Affichez ensuite la liste entière à l'écran.

**Exercice 5.1**

Demandez un nom de pays à l'utilisateur, puis remplacer l'élément numéro 3 de la liste précédente par ce nom. Enfin, affichez la liste entière à l'écran.

**Exercice 5.2**

Créez une liste contenant le nom des 7 jours de la semaine sous forme de strings. Ensuite, demandez à l'utilisateur d'entrer un nombre entier. Le programme affiche alors le nom du jour correspondant à ce nombre (avec la convention que 0 = lundi, 1 = mardi, etc).

## 5.2 Parcourir une liste

Voici un code qui déclare une liste contenant les cinq premiers jours de la semaine puis les affiche un à un.

```
jours = ["lundi", "mardi", "mercredi", "jeudi", "vendredi"]
for i in range(5): #boucle pour parcourir la liste
    print(jours[i])
```

En ligne numéro 2 du code ci-dessus, on a utilisé l'instruction `for i in range(5)` parce qu'il y a 5 éléments dans la liste. Dans la suite de ce cours, nous obtiendrons automatiquement la longueur de la liste grâce à l'instruction `len(ma_liste)`. Grâce à cela, on n'a pas besoin de compter le nombre d'éléments dans la liste à chaque fois que l'on veut parcourir ses éléments.

Il est extrêmement important de comprendre que le code précédent est équivalent à celui-ci :

```
jours = ["lundi", "mardi", "mercredi", "jeudi", "vendredi"]
i = 0
while i < 5: #boucle pour parcourir la liste
    print(jours[i])
    i = i + 1
```

Pour savoir si un élément donné figure au sein d'une liste, on peut utiliser le mot-clé `in`. Par exemple, le code suivant affiche « oui » si le prénom « Alice » figure dans la liste `prenoms`, et « non » sinon.

```python
prenoms = ["Alice", "Bob", "Charles", "Diana", "Earl"]
if "Alice" in prenoms:
    print("oui")
else:
    print("non")
```

Par ailleurs, on peut obtenir l'indice auquel un élément se situe grâce à la méthode `index`. Par exemple :

```python
prenoms = ["Alice", "Bob", "Charles", "Diana", "Earl"]
i = prenoms.index("Charles")
print(i) #affiche 2 car "Charles" est le 3ème élément de la liste
```

Cependant, il est important d'être capable de coder la fonction `index` ou la fonctionnalité `in` soi-même, avant de les utiliser pour gagner du temps.

**Exercice 5.3**

Créez une liste de strings contenant les sept jours de la semaine. Ensuite, grâce à une boucle, affichez un élément sur deux du contenu de cette liste.

**Exercice 5.4**

Créez une liste de strings contenant les sept jours de la semaine. L'utilisateur peut ensuite écrire le nom d'un jour, et le programme lui indique le numéro de ce jour dans la semaine. Attention, le code ne doit pas contenir plus d'une condition (if/else) !

**Exercice 5.5**

Dans le code d'un jeu-vidéo, on écrit le nom des joueurs dans une liste. On écrit également leur meilleur score dans une autre liste, de la façon suivante :

```python
players = ["Alice", "Bob", "Charles", "Diana", "Earl"]
scores = [34, 21.5, 45, 344, 2]
```

Écrivez un code pour afficher le nom de chaque joueur suivi de son score. Le résultat doit ressembler à :
`Alice : 34`
`Bob : 21.5`
et ainsi de suite.

**Exercice 5.6**

Dans un hôpital, on a enregistré les noms des patients dans une liste, et le numéro de leur chambre dans une autre liste. Voici les données en question :

```
patients = ["Alice", "Bob", "Charles", "Diana", "Earl"]
chambres = [3, 2, 4, 2, 5]
```

Écrivez un code qui indique le numéro de la chambre dans laquelle se trouve un patient $x$. Si le patient n'est pas dans la liste, le programme doit afficher « Patient non trouvé ». C'est l'utilisateur qui indique le nom $x$ du patient à chercher.

**Exercice 5.7**

On a inséré les données d'un mini réseau social dans le code suivant :

```
noms = ["Alice", "Bob", "Charles", "Diana", "Earl", "Fab", "Guy"]
ages = [18, 19, 19, 17, 19, 16, 21]
```

Écrivez un code pour afficher l'âge moyen des utilisateurs, ainsi que le nombre d'utilisateurs majeurs. Ce code doit pouvoir fonctionner si l'on modifie les utilisateurs (par exemple, si on en ajoute ou si on en enlève).

**Exercice 5.8 — Difficile**

À l'aide de deux boucles for imbriquées l'une dans l'autre, créez un programme qui affiche tous les couples de lettres possible, avec « A »,« B »,« C »,« D » comme lettres possibles. La réponse sera : AA, AB, AC, AD, BA, etc. Essayez ensuite d'écrire le même code sans utiliser de liste. Qu'observez-vous ?

## 5.3 Manipuler une liste

Pour **ajouter** un élément à la fin d'une liste on utilise l'instruction `append`, comme dans l'exemple ci-dessous :

```
ma_liste = [3,7,1,5]
ma_liste.append(4)
print(ma_liste) #affiche [3,7,1,5,4]
```

À l'inverse, pour **enlever** le dernier élément, on utilise l'instruction `pop`, comme dans l'exemple ci-dessous :

```
ma_liste = [3,7,1,5]
ma_liste.pop()
print(ma_liste) #affiche [3,7,1]
```

### Exercice 5.9

Créez une liste vide et demandez à l'utilisateur d'y ajouter autant de nombres qu'il le souhaite (un par un). Lorsque l'utilisateur écrit « stop », la liste de nombres est affichée et le programme s'arrête.

### Exercice 5.10 — Difficile

Reprenez l'exercice précédent. Une fois que l'utilisateur a fini d'écrire tous les nombres, détectez le plus petit et le plus grand élément de toute la liste, et affichez-les à l'écran.

### Exercice 5.11 — Difficile

Créez un système de création de compte pour un site web imaginaire. Pour cela, créez trois listes vides de la façon suivante :

```
prenoms = []
noms = []
ages = []
```

Lorsqu'un utilisateur écrit « *nouveau compte* », alors il peut entrer son prénom, son nom et son âge, qui seront stockés dans les listes appropriées. Ensuite, le programme lui indique son numéro de compte (ici, ce sera simplement la longueur de la liste à ce moment-là). Lorsque l'utilisateur écrit « *supprimer compte* », alors le dernier compte a avoir été inscrit est supprimé.

### Exercice 5.12 Logiciel de comptabilité — Difficile

Écrivez un logiciel (simpliste) de comptabilité pour un magasin, dont voici les contraintes :

1. L'utilisateur peut écrire "achat", "vente" ou "stop".
2. Si l'utilisateur écrit "achat" ou "vente", alors il doit ensuite indiquer le nom de l'objet acheté puis son prix (chacun via un input différent). Attention, on ne peut pas vendre un objet si l'objet ne fait pas partie de la liste des objets achetés ! En revanche, on peut acheter n'importe quel objet. Vous êtes libres de décider du prix et du nom des différents objets.
3. Tant que l'utilisateur n'écrit pas "stop", les étapes 1 et 2 sont répétées à l'infini.
4. Si l'utilisateur écrit "stop", le programme s'arrête et on affiche :
   (a) Le chiffre d'affaires du jour (c'est-à-dire la somme de toutes les ventes)
   (b) Les dépenses du jour (c'est-à-dire la somme de toutes les achats)
   (c) Le bénéfice du jour (c'est-à-dire le chiffre d'affaires moins les dépenses)
   (d) Le nom de tous les objets vendus

**Pour aller plus loin : Les listes de listes**

Pour représenter des tableaux en plusieurs dimensions, on peut utiliser des listes de listes. Par exemple, au sein d'un jeu-vidéo on pourrait coder une carte en 2D faite de cases grâce à une liste de listes. Les murs peuvent être représentés par le nombre 1 et le vide par le nombre 0. Ainsi, une pièce carrée entourée de murs seraient codée comme :

```
carte = [[1,1,1,1],
         [1,0,0,1],
         [1,0,0,1]
         [1,1,1,1]]
```

Pour obtenir la valeur de la case en coordonée (2 ;3), on écrirait alors `carte[3][2]` (deuxième ligne, troisième colonne).

## 5.4 Itération sur les éléments d'une liste

Jusqu'ici, nous avons parcouru les listes grâce à des boucles (for ou while) qui permettaient d'incrémenter un indice (souvent nommé `i`). Cet indice était à son tour utilisé pour accéder à l'élément numéro `i` de la liste. Bien que cette méthode fonctionne très bien, il peut être pratique, parfois, d'itérer directement sur les éléments de la liste, comme dans l'exemple ci-dessous, où l'on se passe de tout indice !

```
ma_liste = [233, 34, 6, 23.4, 45, 65]
for nombre in ma_liste:
    print(nombre)
```

**Pour aller plus loin : Itérer sur des tuples**

En Python, des **tuples** sont des listes qui ne peuvent pas changer (on dit qu'elles sont **immutables**). Par exemple, `mon_tuple = (1,2,3,4,5)` contient les entiers de 1 à 5. Une liste peut très bien contenir des tuples, par exemple, pour représenter les coordonnées de personnages à l'écran dans un jeu-vidéo. Il est possible d'itérer sur des listes de listes (ou une liste de tuples) de la façon suivante :

```
coords = [(2,3), (10,10), (0,23), (3,4)]
for x,y in coords:
    print(x,y)
```

## 5.5 Accéder à des sous-listes

L'exemple ci-dessous montre comment accéder à des sous-listes spécifiques :

```
ma_liste = [233, 34, 6, 23.4, 45, 65]
print(ma_liste[0:3]) #affiche 233, 34, 6
print(ma_liste[-1]) #affiche le dernier élément (65)
print(ma_liste[3:-1]) #affiche 23.4, 45, 65
print(ma_liste[0:4:2]) #affiche un élément sur deux, de l'indice 0 à 3
print(ma_liste[2:]) #affiche tous les éléments depuis l'indice 2
```

Imaginons que nous voulions accéder à un élément sur deux d'une liste. Voici deux façons de le faire. La première façon est d'utiliser la fonction `range(a,b,delta)`, qui génère les entiers de $a$ à $b$ en incrémentant de *delta* à chaque fois.

```
ma_liste = [233, 34, 6, 23.4, 45, 65]
for i in range(0,len(ma_liste),2): #incrémente de 2 en 2
    print(ma_liste[i])
```

La seconde façon de faire est d'itérer directement sur une sous-liste contenant un élément sur deux :

```
ma_liste = [233, 34, 6, 23.4, 45, 65]
for nombre in ma_liste[0:-1:2]:
    print(nombre)
```

# Chapitre 6

# Les fonctions

Lorsqu'un bout de code est répété plusieurs fois dans un programme, il est préférable de ne pas le réécrire à chaque fois. À la place, utiliser une fonction est souvent plus approprié. Dans l'exemple suivant, les lignes de 2 à 8 sont identiques aux lignes de 11 à 17 :

```
majorite = 18
age = int(input("Quel est votre âge ?"))
if age >= majorite:
    print("Vous êtes majeur")
else:
    print("Vous serez majeur dans " + (18 - age) + " ans")
majorite = 21
print("Maintenant, l'âge de la majorité a changé")
age = age = int(input("Quel est votre âge ?"))
if age >= majorite:
    print("Vous êtes majeur")
else:
    print("Vous serez majeur dans " + (majorite - age) + " ans")
```

À la place de répéter ces lignes, on peut les « ranger » dans une fonction [1]. Voici comment cela se présente :

```
def traiter_age(): #début de la fonction
    age = int(input("Quel est votre âge ?"))
    if age >= majorite:
        print("Vous êtes majeur")
    else:
        print("Vous serez majeur dans " + (18 - age) + " ans")
majorite = 18
traiter_age() #premier appel de la fonction
```

1. Dans certains langages, on parle de « procédure » plutôt que de « fonction » si cette dernière ne retourne pas de valeur (voir plus loin). En Python, les deux termes sont utilisés de façon interchangeable.

```
print("Maintenant, l'âge de la majorité a changé")
majorite = 21
traiter_age() #second appel de la fonction
```

Pour résumer, les fonctions permettent : 1) de réduire les erreurs d'écriture du code ; 2) de rendre le code plus compact et plus lisible.

Il est important de comprendre que le code à l'intérieur d'une fonction n'est pas exécuté tant que la fonction n'est pas **appelée**, même si la fonction est **définie** avant ! Dans l'exemple précédent, la fonction est définie de la ligne 1 à la ligne 6, mais son code est exécuté en ligne 8 une première fois puis en ligne 11 une seconde fois.

**Exercice 6.0**

Réécrivez le code suivant en créant une fonction pour éviter de répéter certaines instructions.

```
a = 3
x = 0
while x != a:
    x = int(input("Devinez le nombre (entre 1 et 10)"))
print("Première partie gagnée")
a = 9
x = 0
while x != a:
    x = int(input("Devinez le nombre (entre 1 et 10)"))
print("Seconde partie gagnée")
a = 2
x = 0
while x != a:
    x = int(input("Devinez le nombre (entre 1 et 10)"))
print("Troisième partie gagnée")
```

## 6.1 Les arguments d'une fonction

On peut affiner le comportement d'une fonction grâce à des paramètres. Prenons l'exemple d'une fonction qui convertit des francs Suisses en Euros et affiche cette valeur. Cette fonction agit sur un nombre (la somme en francs Suisse) : on dit qu'elle **prend** cette somme en argument. Le ou les arguments pris par une fonction sont indiqués dans les parenthèses au moment de la définition :

```
def conversion(somme_CHF): #prend l'argument somme_CHF
  taux = 1.2 #taux du jour
  somme_EUR = somme_CHF / taux
  print(somme_CHF + " CHF = " + somme_EUR + " €")
```

```
5
6 conversion(3) #on passe le nombre 3 en argument
7 conversion(254.5) #on passe le nombre 254.5 en argument
```

Pour passer plusieurs arguments à une fonction, on les sépare à l'aide d'une virgule. Voici une adaptation de l'exemple précédent où le taux de conversion peut être spécifié.

```
1 def conversion(somme_CHF, taux): #prend l'argument somme_CHF
2   somme_EUR = somme_CHF / taux
3   print(somme_CHF + " CHF = " + somme_EUR + " €")
4
5 conversion(3, 1.2) #somme_CHF = 3 et taux = 1.2
6 conversion(3, 1.05) #Cette fois, taux = 1.05
```

**Exercice 6.1**

Écrivez une fonction f(x) qui prend un nombre en argument et affiche son carré. Testez-la sur les appels suivants :

```
f(3) #doit afficher 9
f(-2) #doit afficher 4
```

**Exercice 6.2**

Écrivez une fonction billet_TPG(zone, age). Si l'argument zone vaut 1, le prix est de 4 CHF. Si zone vaut 2, c'est un saut de puce et le prix du billet est de 2 CHF. Si l'age est inférieur à 18 et supérieur ou égal à 8, le prix est divisé par deux. Si l'age est inférieur à 8, le billet est gratuit. La fonction doit afficher le prix final. Si la zone n'est pas 1 ou 2, ou si l'âge est négatif, un message d'erreur doit être affiché.

```
billet_TPG(1,56) #doit afficher 4 CHF
billet_TPG(1,12) #doit afficher 2 CHF
billet_TPG(1,2) #doit afficher 0 CHF
billet_TPG(2,18) #doit afficher 2 CHF
billet_TPG(2,17) #doit afficher 1 CHF
billet_TPG(3,17) #doit afficher un message d'erreur
billet_TPG(1,-2) #doit afficher un message d'erreur
```

**Pour aller plus loin : portée des variables**

Les variables qui sont déclarées avec le mot-clé  à l'intérieur d'une fonction n'existeront pas en-dehors de cette fonction. En revanche, les variables qui sont déclarées sans mot-clé sont dites « globales » : elles existent « partout » une fois qu'elles sont déclarées.

Cette **portée** des variables est également applicable à tous les bouts de code délimités par des accolades : les boucles et les conditions suivent donc la même règle. **Il est en général préférable de déclarer les variables avec le mot-clé** pour éviter que les variables déclarées dans une fonction, dans une boucle ou dans une condition ne soient confondues avec d'autres variables déclarées ailleurs mais portant le même nom !

## 6.2 Valeur de retour

Jusqu'ici, nous n'avons utilisé des fonctions que pour afficher des valeurs. Une fonction peut également **retourner** une valeur grâce au mot-clé `return`. En voici un exemple d'utilisation :

```
def f(x): #f(x) retourne le cube de x
    return x*x*x
a = f(2)
b = f(3)
c = f(-2)
print(a) #affiche 8
print(b) #affiche 27
print(c) #affiche -8
```

Plusieurs lignes d'une même fonction peuvent retourner une valeur :

```
def mineur_ou_majeur(x):
    if x < 18:
        return "mineur"
    else:
        return "majeur"
a = mineur_ou_majeur(19)
b = mineur_ou_majeur(7)
print(a) #affiche "majeur"
print(b) #affiche "mineur"
```

Une fois que la ligne contenant l'instruction de retour est atteinte, l'exécution de la fonction est stoppée et les lignes suivantes sont ignorées. Par conséquent, la fonction précédente est équivalente à :

```
def mineur_ou_majeur(x):
    if x < 18:
        return "mineur" #on quitte la fonction si on arrive ici
    return "majeur" #ne peut être atteint que si x >= 18
```

### Exercice 6.3

Modifiez la fonction `billet_TPG` écrite dans l'exercice plus haut afin qu'elle retourne le prix du billet.

```
prix = billet_TPG(1,56)
print(prix) #doit afficher 4
```

### Exercice 6.4

Ecrivez une fonction qui implémente la fonction mathématique $f(x) = 3 \cdot x - 6$.

```
print(f(2)) #doit afficher 0
print(f(0)) #doit afficher -6
```

### Exercice 6.5

Écrivez une fonction qui implémente la fonction mathématique $f(x) = \frac{1}{x}$. Si la valeur de `x` est zéro, un message d'erreur doit être affiché : « x=0 est hors du domaine de définition ! »

### Exercice 6.6

Écrivez une fonction $multiples3(a, b)$ qui retourne une liste contenant tous les multiples de 3 compris entre $a$ et $b$.

### Exercice 6.7

Écrivez une fonction $est_premier(x)$ qui retourne le booléen `True` si $x$ est un nombre premier, et qui retourne `False` sinon.

### Exercice 6.8

En réutilisant la fonction de l'exercice précédent, écrivez une fonction nommée $nb_premiers(a, b)$ qui retourne une liste contenant tous les nombres premiers compris entre $a$ et $b$.

### Exercice 6.9

En mathématiques, la fonction factorielle est une fonction $f(x)$ qui retourne la multiplication de tous les entiers de 1 jusqu'à x. Par exemple, $f(4) = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. À l'aide d'une fonction qui s'appelle elle-même, implémentez la fonction factorielle (sans utiliser de boucle while ni de boucle for !).

# Chapitre 7

# Manipulation des chaînes de caractères

Les chaînes de caractères, déjà abordées dans le chapitre sur les variables (chapitre 2), peuvent être vues comme des listes de caractères (d'où leur nom). Ainsi, la plupart des méthodes de manipulation des listes s'appliquent également pour les strings ! Il existe cependant une différence importante : les strings sont mutables (voir 5.4), c'est-à-dire qu'on ne peut jamais modifier un string une fois qu'il a été déclaré. Dans un tel cas, on utilisera plutôt une copie de ce string.

Pour obtenir la longueur d'une chaîne, on utilise la fonction len() :

```
x = "Bonjour"
n = len(x) # n contient maintenant la valeur 7
```

Pour savoir si une chaîne contient un caractère ou une suite de caractères, on utilise le mot-clé `in` :

```
x = "Bonjour"
if "o" in x:
        print(x, "contient au moins un o")
if "our" in x:
        print(x, "contient au moins un our")
```

Pour compter le nombre d'apparitions d'un caractère ou d'une suite de caractères à l'intérieur d'une chaîne, on utilise la méthode count() :

```
x = "Bonjour"
n_o = x.count("o") # n_o vaut 2
n_r = x.count("r") # n_r vaut 1
n_z = x.count("z") # n_z vaut 0
```

Pour obtenir une copie d'un string où chaque **occurence** de `x` est remplacée par `y`, on utilise `replace(x,y)` :

```
mot1 = "Bonjour"
mot2 = mot1.replace("o", "e") #mot2 contient "Benjeur"
```

Il existe encore bien d'autres fonctionnalités que nous n'abordons pas ici, mais qui sont disponibles dans la documentation de Python sur le web.

**Exercice 7.0**

Créez une liste de strings nommée `liste_des_mots` contenant les mots suivants : "exceptionnel", "electronique", "pomme", "mangue", "partez", "mangez", "carotte", "pois", "la", "kiwi". Cette liste sera utilisée dans certains des exercices suivants.
Affichez la longueur de la liste dans la console.

**Exercice 7.1**

Affichez la longueur de chaque mot contenu dans `liste_des_mots`

**Exercice 7.2**

Pour chaque mot dans `liste_des_mots`, affichez le nombre de "e" qu'il contient.

**Exercice 7.3**

Pour chaque mot dans `liste_des_mots`, si ce mot contient un "a" et ne contient pas de "z", affichez sa longueur.

**Exercice 7.4**

Pour chaque mot dans `liste_des_mots`, si ce mot contient un "z", affichez le nombre d'apparitions de chacun des caractères du mot.

**Exercice 7.5**

Pour chaque mot dans `liste_des_mots`, affichez son score. Le score d'un mot est donné par la formule suivante :

$$score = 10(nombre_de_x + nombre_de_y + 0.3 \cdot nombre_de_z)$$

**Exercice 7.6**

Pour chaque mot dans `liste_des_mots`, affichez son score. Le score d'un mot est la somme du score de chacune de ses lettres. Les voyelles comptent pour 1 point, sauf le "y". Les consonnes comptent pour deux points, sauf le "y" et le "x" qui comptent pour 3, et le "k" et "w" qui comptent pour 4.

### Exercice 7.7

Pour chaque mot dans `liste_des_mots`, affichez chacune de ses lettres en indiquant son score (voir exercice 6) entre parenthèses. Si l'un de ces scores est supérieur à deux, affichez également "joli!". Par exemple, pour le mot, "kiwi", l'output doit être :

```
k (4) joli !
i (1)
w (4) joli !
i (1)
```

### Exercice 7.8 – morse (Hors programme)

Créez une application qui convertit un texte en morse. L'input de l'utilisateur peut contenir les 26 lettres latines majuscules (sans accents) ainsi que l'espace vide. Veuillez vous documenter sur l'utilisation du Morse sur Wikipédia : `https://fr.wikipedia.org/wiki/Code_Morse_international`. Il est conseillé d'utiliser un **dictionnaire Python** pour réaliser cet exercice (cf. chapitre correspondant). Dans cet exercice, vous utiliserez les espaces vides pour délimiter les lettres. Par exemple, si l'input de l'utilisateur est « chat », alors l'output doit être « -.-. .... .- - ».

### Exercice 7.9 – Scrabble (Hors programme, travail de longue haleine)

Cherchez les règles du Scrabble sur internet et implémentez le score exact d'un mot donné par l'utilisateur (en imaginant que c'est le premier mot à être posé dans le jeu). Le mot peut être donné en input par l'utilisateur.

### Exercice 7.10 – Pendu (Hors programme, difficile !)

Programmez un jeu du pendu où l'utilisateur doit deviner le mot. Le mot est choisi au hasard parmi ceux que vous définirez en tant que programmeur. Plutôt qu'un dessin, une étoile est affichée à l'écran pour chaque faute. Au bout de 8 fautes, le joueur a perdu.

## Exercice 7.11 — Fouiller dans le dictionnaire

Un petit dictionnaire des mots français (sans accents) est mis à disposition sur Classroom sous le nom de fichier `french.txt` (source : freelang.com).
Voici comment générer la liste des mots en Python (il faut absolument que le fichier des mots soit dans le même dossier que le script Python) :

```
with open("french.txt","r") as f:
    words = f.readline().split(",")
```

A) Affichez tous les termes de plus de 15 caractères.
B) Affichez tous les termes de moins de 3 caractères.
C) Affichez le plus long « terme » de ce dictionnaire.
D) Affichez tous les termes qui contiennent deux $k$.

## Exercice 7.12 — Faux Scrabble

A) Le but de cet exercice est de trouver le mot de la langue française qui, en théorie, donnerait le meilleur score, si l'on associe à chaque lettre de l'alphabet un score qui correspond à son numéro dans l'alphabet : $a = 1$, $b = 2$, ..., $z = 26$. Par exemple, le mot « la » vaut $12 + 1 = 13$ points. Tous les caractères qui ne sont pas des lettres (par exemple les tirets, comme dans « rendez-vous » ou les espaces, comme dans « blanc d'oeuf ») comptent pour 0 points.

B) Quel est le mot avec le plus petit score ?

C) Quel est le plus long mot de ce dictionnaire ?

## Exercice 7.13 — Hacking

Trouvez le mot de passe de Bob, sachant que son mot de passe fait 9 caractères de long, que la troisième lettre est un « c » et que la dernière lettre est un « i ».

## Exercice 7.14 — Mots-croisables

Trouvez la liste des couples de mots qui peuvent être « croisés » (comme dans un mot-croisés, un mot vertical et un mot horizontal). Il faut que la lettre de croisement soit la dixième lettre de chaque mot. Le premier mot doit contenir un « z » et le second mot doit contenir un « k ».

## Exercice 7.15 — Mots sans répétitions

A)Trouvez le nombre de mots du dictionnaire qui ne comportent que des caractères différents, c'est-à-dire dont aucun caractère n'est répété. Par exemple, « zone » ne possède pas de caractère répété, tandis que « zoo » possède un caractère répété.

B) Quel est le plus long mot sans caractère répété ?

### Exercice 7.16 — Palindromes

A) Affichez tous les termes qui sont des palindromes. Exemples de palindromes : « radar », « solos ».

B) Quel est le plus long palindrome ?

# Chapitre 8

# Nombres pseudo-aléatoires

Dans biens des applications (cryptographie, jeux-vidéos, sciences, ...) il est très pratique de pouvoir utiliser des nombres aléatoires. Par exemple, pour placer un millier d'arbres dans une forêt d'un jeu-vidéo, il est préférable d'utiliser l'aléatoire que de choisir manuellement l'emplacement exact de chaque arbre.

Dans tout ce chapitre, nous entendrons « pseudo-aléatoire » à chaque fois que nous dirons « aléatoire » : en effet, la méthode par défaut de génération des nombres aléatoires utilisés par la plupart des machines modernes [1] est déterministe : en partant d'une valeur initiale appelée « seed » (graine), la suite des nombres aléatoires générés par l'algorithme est entièrement déterminée et reproductible (ce qui est bien pratique en sciences ou encore dans des jeux où la génération du monde doit être à la fois aléatoire et reproductible (*e.g.* Minecraft).

En Python, le module built-in [2] `random` contient de nombreuses fonctions pour effectuer un traitement aléatoire des données. Nous n'en voyons ici que deux. Dans tous les codes qui suivent, nous supposerons que la première ligne contient l'instruction :

```
import random
```

## 8.1 Génération d'un `int` aléatoire

Pour générer un nombre entier aléatoire entre deux bornes, il existe la fonction `randint` :

```
a = random.randint(-5, 5) #a contient mainteant un nombre entre -5 et 5
b = random.randint(0, 100) #b contient mainteant un nombre entre 0 et 100
```

1. il existe des méthodes basées sur des processus physiques réellement aléatoires (dans la mesure d'un cadre philosophique donné), mais nous n'en parlons pas ici.
2. c'est-à-dire présent par défaut avec votre version de Python et n'ayant donc pas besoin d'être installé.

```
print(a, b)
```

Il est très important de comprendre que :

1. La valeur contenue dans les variables aléatoires ne peut pas être connue avant l'exécution du code.
2. Si l'on exécute le code plusieurs fois, il est probable que le résultat soit différent à chaque fois.

**Exercice 8.0**

Demandez deux nombres entiers à l'utilisateur, puis affichez un nombre entier au hasard compris entre ces deux nombres.

**Exercice 8.1**

Tirez dix nombres au hasard entre 1 et 10, puis affichez leur somme.

**Exercice 8.2**

Tirez mille nombres au hasard entre 1 et 100, puis affichez leur moyenne.

**Exercice 8.3**

Générez une « forêt » de coordonnées en 2D. Pour cela, affichez dans la console 100 coordonnées $(x; y)$ tirées au hasard. Chaque coordonnée représente un pixel de l'écran. La valeur $x$ doit être comprise entre 0 et 1920 et la valeur $y$ entre 0 et 1080 (écran HD).

**Exercice 8.4 – Choix au hasard dans une liste**

Déclarez une liste de strings contenant 5 prénoms. Ensuite, affichez l'un de ces prénoms au hasard (chaque prénom doit avoir autant de chances que les autres d'apparaître).

**Pour aller plus loin : choix d'un élément au hasard au sein d'une liste**

Pour choisir un élément au hasard au sein d'une liste, on peut utiliser la fonction `choice` du module `random` :

```
couleurs = ["rouge", "vert", "bleu", "jaune", "noir"]
couleur_hasard = random.choice(couleur) #choisit une couleur au hasard
print(couleur_hasard)
```

**Exercice 8.5 – Jets de dés**

Proposez à l'utilisateur de jeter un dé virtuel à 6 faces (s'il écrit « ok », on jette le dé, sinon on quitte l'application). Tant que le dé ne tombe pas sur la valeur 3, l'utilisateur peut continuer de jouer. Dès qu'il tombe sur 3, la partie est finie et le score du joueur est affiché. Le score vaut la somme des valeurs obtenues par le joueur.

**Exercice 8.6 – Générateur de phrases**

Déclarez trois listes. Une liste contient des sujets de phrases au singulier (« Je », « Tu », « Il », « Alice », etc...). Une seconde liste contient des verbes et un éventuel complément (« mange », « danse avec », « déteste », etc...). Une troisième liste contient des compléments de phrases (« les pommes », « les carottes », « les insectes », etc...). Générez 10 phrases aléatoires à partir de ces listes.

**Exercice 8.7 – Statistiques sur les dés**

Simulez 1 million de jets de dé à 6 faces. Quelle est la moyenne de la valeur obtenue ? Quel pourcentage des lancers obtient-on la valeur 6 ? Quel pourcentage des lancers obtient-on exactement trois fois la valeur 6 de suite ?

**Exercice 8.8 – Statistiques sur les dés – Version difficile !**

Simulez 1 million de jets de dé à 6 faces. Combien de fois obtient-t-on trois 6 à la suite ? Quel est environ le pourcentage de chance d'obtenir un triplet d'après cette expérience ? Y a-t-il une fois où l'on obtient dix 6 à la suite ?

## 8.2 Génération d'un `float` aléatoire

Pour générer un nombre aléatoire entre 0 et 1, il existe la fonction `random` :

```
a = random.random() #a contient mainteant un nombre entre 0 et 1
print(a)
```

**Exercice 8.9**

Affichez un float aléatoire entre 0 et 100

**Exercice 8.10**

Affichez un float aléatoire entre 3 et 8

### Exercice 8.11

Affichez un float aléatoire entre -6 et 8

### Exercice 8.12

Demandez deux nombres entiers à l'utilisateur, puis affichez un float au hasard compris entre ces deux nombres.

### Exercice 8.13 – Approximation $\pi$ – Difficile !

Tirez un millions de paires de float aléatoires entre 0 et 1. Si ces nombres constituent des coordonnées, combien se situent à une distance plus petite que 1 de l'origine ? Déduisez-en la valeur approximative de $\pi$.

# Chapitre 9

# Pour aller plus loin

Les concepts présentés dans ce chapitre ne figurent pas au programme. Ils peuvent toutefois s'avérer très utiles pour la suite de votre apprentissage de la programmation en Python. Nous n'en donnons ici qu'un aperçu extrêmement simplifié et superficiel ; le but est simplement de vous donner une idée de ce qu'il est possible de faire avec ces différents outils, afin que vous sachiez qu'ils existent et que vous puissiez les utiliser si vous en avez besoin, après avoir davantage approfondi le sujet en utilisant des ressources externes.

## 9.1 Les dictionnaires

Dans un précédent chapitre, nous avons vu comment utiliser des listes en Python. Si les listes sont utiles pour toute une famille de problèmes, il en existe certains pour lesquels elle ne sont pas bien adaptées. Prenons l'exemple suivant : on veut constituer un lexique de traductions de mots du français à l'anglais. Avec des listes, on pourrait avoir un code du type :

```
french = ["chat", "voiture", "deux", "rouge"]
english = ["cat", "car", "two", "red"]
```

Cependant, cette approche comporte deux défauts majeurs. Le premier est que l'ordre des mots d'une liste doit toujours concorder avec ceux de l'autre. La moindre erreur rend tout le reste du lexique faux ! Le second défaut est qu'il n'est pas aisé de récupérer la traduction d'un mot donné, car il faut d'abord repérer l'indice de ce mot au sein de sa liste, avant d'aller chercher le mot de même indice dans la liste de l'autre langue.

Une approche utilisant un **dictionnaire** s'avère alors plus simple à utiliser, car un dictionnaire sert précisément à **faire le lien entre des clés et des valeurs** :

```
traduction = {"chat":"cat", "voiture":"car", "deux":"two", "rouge":"red"}
```

Pour récupérer la valeur associée à la clé « voiture », par exemple, il suffit d'y accéder en utilisant des crochets :

```
traduction = {"chat":"cat", "voiture":"car", "deux":"two", "rouge":"red"}
trad_voiture = traduction["voiture"]
print(trad_voiture) #affiche "car"
```

D'une façon générale, on peut maintenant accéder à n'importe quelle clé demandée par l'utilisateur.

Dans le cas où l'on essaie d'accéder à une clé qui n'existe pas dans le dictionnaire, Python génère une erreur. Il est donc important de gérer ce genre de cas. Deux approches sont possibles. La première est de vérifier si la clé existe dans le dictionnaire :

```
traduction = {"chat":"cat", "voiture":"car", "deux":"two", "rouge":"red"}
mot = input("Quel mot voulez-vous traduire?")
#ici, pas de garantie que le mot soit dans notre lexique
if mot in traduction:
    print("Votre mot se traduit:", traduction[mot])
else:
    print("Votre mot n'est pas dans le lexique")
```

Une autre approche, souvent plus utile, est de faire appel à la méthode `get` des dictionnaires, qui peut prendre deux arguments : la clé à chercher et la valeur à retourner en cas de clé non trouvée. Voici un exemple :

```
traduction = {"chat":"cat", "voiture":"car", "deux":"two", "rouge":"red"}
mot = input("Quel mot voulez-vous traduire?")
trad_mot = traduction.get(mot, "(???)")
#si la traduction n'existe pas, on affichera des points d'interrogation
print("Traduction de votre mot:", trad_mot)
```

La méthode `get` peut même être utilisée avec un seul argument (la clé) ; dans ce cas, si la clé n'est pas trouvée, `get` retourne la valeur `None` (un type de variable qui permet de refléter le concept de « rien »).

Il est important de noter que, tout comme les valeurs au sein des listes, les valeurs stockées dans le dictionnaire peuvent être de n'importe quel type. Par exemple :

```
prix = {"Pomme":0.5, "Poire":1, "Cerises":2, "Mangue":5}
print(prix.get("Cerises")) #ceci affiche "2"
print(prix.get("Framboises")) #ceci affiche "None"
```

Enfin, il est possible d'itérer sur les clés d'un dictionnaire comme on le ferait sur les valeurs d'une liste :

```python
prix = {"Pomme":0.5, "Poire":1, "Cerises":2, "Mangue":5}
for fruit in prix:
    print("L'article", fruit, "coûte", prix[fruit], "CHF")
```

Notons qu'en Python ainsi que dans d'autres langages tels que Javascript, la structure des dictionnaire est si importante qu'elle est en fait utilisée pour représentée des classes d'objets (sujet du prochain chapitre). Par exemple, le code suivant permet de simuler un type d'objet `Livre` pour la gestion d'une bibliothèque, où l'on s'en sert pour afficher automatiquement tous les livres écrits en français :

```python
harry_potter = {"titre":"Harry Potter", "auteur":"J.K. Rowling",
                "pages":300, "langue":"Anglais"}
les_miserables = {"titre":"Les Misérables", "auteur":"V. Hugo",
                "pages":900, "langue":"Français"}
belle_du_seigneur = {"titre":"Belle du Seigneur", "auteur":"A. Cohen",
                "pages":800, "langue":"Français"}

tous_les_livres = [harry_potter, les_miserables, belle_du_seigneur]

for livre in tous_les_livres:
    if livre["langue"] == "Français":
        print("Le livre en français", livre["titre"],
              "a été écrit par", livre["auteur"])
```

### Exercice 9.0 - Le magasin de bonbons

Créez un dictionnaire pour représenter l'inventaire d'un magasin de bonbons. Les clés sont les types de bonbons et les valeurs sont les nombres en stock. Écrivez une fonction pour afficher le stock d'un bonbon spécifique et une autre fonction pour "acheter" des bonbons, qui réduit le stock.

**Exemple d'output :**

```
Stock avant achat : { "chocolat" : 20, "bonbon" : 15 }
Achat de 5 chocolats.
Stock après achat : { "chocolat" : 15, "bonbon" : 15 }
```

### Exercice 9.1 - Jeu des cris d'animaux

Construisez un dictionnaire où chaque clé est un animal et la valeur associée est le son que l'animal fait. Demandez à l'utilisateur de deviner le son d'un animal choisi au hasard. Vérifiez si la réponse est correcte.

**Exemple d'output :**

```
Quel son fait le chat ?
Votre réponse : miaou
Correct !
```

**Exercice 9.2 - Enquête de popularité**

Gérez un dictionnaire pour enregistrer les résultats d'une enquête sur les films préférés. Les clés sont les noms des films et les valeurs sont le nombre de votes. Ajoutez des fonctions pour voter et pour afficher le film le plus populaire.

**Exemple d'output :**

```
Votes actuels : { "Inception" : 10, "Matrix" : 8 }
Vote pour Inception.
Nouveau total : { "Inception" : 11, "Matrix" : 8 }
Le film le plus populaire est Inception.
```

**Exercice 9.3 - Le dictionnaire des synonymes**

Créez un dictionnaire des synonymes où chaque clé est un mot et chaque valeur est une liste de synonymes. Proposez une interface pour que l'utilisateur puisse obtenir les synonymes d'un mot donné.

**Exemple d'output :**

```
Entrez un mot : grand
Synonymes : large, vaste, élevé
```

**Exercice 9.4 - Suivi des étudiants**

Utilisez un dictionnaire pour suivre les notes des étudiants dans différents cours. Les clés sont les noms des étudiants et les valeurs sont d'autres dictionnaires contenant les noms des cours et les notes. Ajoutez des fonctions pour ajouter des notes, calculer la moyenne des notes de chaque étudiant, et afficher les résultats.

**Exemple d'output :**

```
Notes de Alice : { "Maths" : 5, "Physique" : 4 }
Moyenne de Alice : 4.5
```

**Exercice 9.5 - Transformation en liste**

Ecrivez un code qui transforme les couples clé-valeur d'un dictionnaire en une liste de tuples à 2 éléments, puis affichez le minimum ces valeurs. Les clés sont des strings et les valeurs des nombres.

**Exemple d'output :**

```
dictionnaire : { "Maths" : 5, "Physique" : 4 }
réponse : [("Maths", 5), ("Physique", 4)] minimum : 4 en Physique
```

## 9.2 Les classes

Nous touchons ici à un sujet très sensible au sein des communautés de développeurs. Par ailleurs, le sujet des classes et de leur utilisation au sein de l'architecture des logiciels est un sujet très vaste, au sujet duquel de nombreux ouvrages ont été écrits. Nous nous contenterons ici de donner une introduction très superficielle à ce concept, sans entrer

dans les détails. Par ailleurs, dans le cadre de ce cours qui s'adresse aux débutants, nous recommandons d'utiliser les classes en tant que « regroupeurs des données », et non en tant qu'objets qui servent à implémenter des concepts de programmation orientée objet (cf. plus bas). Notre utilisation des classes se cantonne à une structuration du code qui permet de simplifier la vie du programmeur. Cependant, de nombreuses personnes sur le Web ne partageront pas cet avis [1], y compris au sein de cours prestigieux ; il ne faut donc pas s'étonner de la disparité d'opinions et de pratiques sur le sujet.

Dans le paysage général de la programmation et du génie logiciel, les classes sont souvent associées aux langages dits *orientés objets*. Cependant, il est tout à fait possible, comme en Python, d'utiliser des classes en tant que simples structures de données, sans adhérer à la philosophie de la programmation orientée objet. Il serait d'ailleurs impossible d'y adhérer parfaitement en Python, étant donné que le langage ne permet pas de restreindre l'accès aux attributs d'une classe (concepts de *private* et *public* et d'*encapsulation*).

Par ailleurs, les classes sont souvent associées au concept d'héritage, que nous ne couvrons pas ici. D'une façon générale, nous conseillons vivement de limiter l'héritage de classes autant que possible car cela rend le code plus difficile à comprendre et à maintenir, à moins que l'on ait minutieusement réfléchi à l'architecture du code, et ce dans les types de problèmes qui s'y prêtent. L'héritage multiple en revanche est à éviter tout à fait, car il rend le code très difficile à comprendre.

### 9.2.1 Définition d'une classe

Une classe est une structure de données qui permet de regrouper des données et des fonctions qui agissent sur ces données, que l'on appelera des **méthodes**. Voici un exemple simple de classe en Python :

```
class Personnage:
    def __init__(self, nom, age):
        self.nom = nom #'self' sert à désigner l'objet dont on parle
        self.age = age
```

Ce code définit en quelque sorte le « chablon » de ce que doit être un personnage dans notre code. Pour créer un personnage, il suffit d'**instancier** la classe `Personnage` en lui passant un nom et un âge :

```
p1 = Personnage("Alice", 25) #p1 correspond à Alice, 25 ans
p2 = Personnage("Bob", 30) #p2 correspond à Bobe, 30 ans
```

Notons que la classe `Personnage` qui possède deux **attributs**, `nom` et `age`. L'attribut `nom` est initialisé avec la valeur de la variable `nom` passée en argument lors de la création d'une

1. Cela concerne en particulier les cours écrits dans les années 90-2000, où le paradigme orienté objet était particulièrement populaire. Le langage Java est l'emblème de cette époque et demeure extrêmement utilisé dans les entreprises en raison de la popularité qu'il a connu à cette époque.

instance de la classe, et de même pour l'attribut `age`. La méthode `__init__(self, ...)` est une méthode spéciale en Python qui est appelée lors de la création d'une instance de la classe. Elle est souvent utilisée pour initialiser les attributs de la classe.

Dans ce cas, les classes sont utiles car cela nous évite de devoir maintenir deux listes séparées pour les noms et les âges des personnages. De plus, cela nous permet de regrouper les données et les fonctions qui agissent sur ces données, ce qui rend le code plus lisible et plus facile à maintenir. En effet, on peut définir des fonctions appelées « méthodes » qui agissent sur les données de la classe. Par exemple, on pourrait ajouter une méthode `decrire` à notre classe `Personnage` pour afficher de manière personnalisée les informations sur un personnage :

```
class Personnage:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def decrire(self): #les méthodes ont toujours "self" en premier arg.
        print("Je me nomme", self.nom, "et j'ai", self.age, "ans.")

p1 = Personnage("Alice", 25)
p1.decrire() #affiche "Je me nomme Alice et j'ai 25 ans."
```

Tout comme les fonctions, les méthodes peuvent prendre des arguments et retourner des valeurs. Dans le code ci-dessous, on définit une méthode nommée `calcul_age(annee)` qui prend en argument une année et retourne l'âge du personnage lors cette année-là :

```
class Personnage:
    def __init__(self, nom, age_en_2024):
        self.nom = nom
        self.age = age_en_2024

    def calcul_age(self, annee):
        return annee - 2024 + self.age

p1 = Personnage("Alice", 25)
age_en_2030 = p1.calcul_age(2030)
print("Alice aura", age_en_2030, "ans en 2030.")
```

## 9.3 Les modules et les packages

Il est en général bon de structurer les projets en plusieurs fichiers ou dossiers afin de séparer clairement la responsabilité des différentes parties du code. Par exemple, dans un jeu vidéo, on pourrait trouver un fichier nommé `main.py` qu'il faut lancer pour lancer le jeu, ainsi qu'un fichier `personnage.py` qui contient la définition des personnages du jeu, et

un fichier `levels.py` qui contient la définition des niveaux du jeu. Au sein de `main.py`, on pourrait importer la classe `Personnage` du fichier `personnage.py` de la manière suivante :

```
from personnage import Personnage
```

Il est également possible d'importer toutes les classes et fonctions d'un fichier en utilisant l'étoile, mais nous décourageons cette pratique car cela empêche de savoir d'où vient quel objet, ce qui peut mener à des confusions, surtout dans les grands projets.

```
from personnage import *
```

Enfin, on peut simplement importer le module entier :

```
import personnage
```

Dans ce cas, on doit préciser le nom du module suivi d'un point pour accéder à ses classes et fonctions :

```
p1 = personnage.Personnage("Alice", 25)
```

Tout ce que l'on vient de dire ne fonctionne que si les fichiers figurent dans le même répertoire. La gestion des modules est un sujet complexe qui dépasse le cadre de ce cours, nous donnons ici les bases pour que vous puissiez commencer à structurer vos projets de manière plus claire.

Enfin, il faut savoir que vous avez déjà utilisés des modules externes. Pour être plus pécis, à chaque fois que nous avons utilisé des librairies telles que math, random ou turtle, nous avons importé des modules ou bien des **packages**, qui sont simplement des collections de plusieurs modules. Il est possible d'installer des packages externes en utilisant le gestionnaire de paquets pip, qui est installé par défaut avec Python. Par exemple, pour installer le module `pygame`, il suffit de taper la commande suivante dans un terminal :

```
pip install pygame
```

Une fois le module installé, vous pouvez l'importer dans votre code comme s'il s'agissait d'un module standard, dans le même dossier que votre code.

**Exercice 9.6 - Création de module simple**

Créez un module Python nommé `utils.py`. Dans ce module, définissez une fonction `saluer()` qui prend un nom en paramètre et retourne une salutation, "Bonjour {nom}". Ensuite, dans un fichier `main.py`, importez ce module et utilisez la fonction `saluer()` pour afficher une salutation.

**Exercice 9.7 - Structurer un package**

Organisez un petit package Python nommé `jeu`. Ce package doit contenir deux modules : `personnages.py` et `niveaux.py`. Le module `personnages.py` doit contenir une classe `Personnage` avec des attributs comme `nom` et `force`. Le module `niveaux.py` doit définir une classe `Niveau` avec des attributs comme `difficulte` et `monstres`. Assurez-vous d'inclure un fichier `__init__.py` (laissez-le vide) pour que Python traite le dossier comme un package. Dans un fichier `main.py` externe au package, importez et utilisez les classes définies.

## 9.4 Gestion des environnements avec venv

Lorsque vous travaillez sur des projets Python plus importants, surtout en collaboration, il est crucial de maintenir l'isolation de l'environnement de votre projet pour éviter les conflits entre les dépendances. Python offre un outil intégré nommé `venv`, qui permet de créer des environnements virtuels. Voici comment créer et activer un environnement virtuel sous Windows :

```
python -m venv mon_projet_env
mon_projet_env\Scripts\activate
```

Une fois l'environnement activé, toutes les installations de packages effectuées avec `pip` seront limitées à cet environnement, sans affecter les autres projets ou le système global.

## 9.5 Les graphiques avec Matplotlib

Matplotlib est une librairie (package) populaire qui permet de créer des graphiques et toutes sortes de visualisations en Python. Voici un exemple simple de code qui permet de visualiser des données $y$ en fonction de données $x$ :

```python
import matplotlib.pyplot as plt

# Dans cet exemple, on écrit directement les coordonnées des points
x = [1, 3, 4, 5, 7]
y = [1, 2, -3, 5, 2]

# Utilisation de Matplotlib
plt.plot(x, y, "o-")
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

De même, il est aisé de tracer des fonctions mathématiques. Par exemple, pour tracer la fonction $f(x) = (x-3)^2 + 1$ sur l'intervalle $[-5, 10]$ :

```
import matplotlib.pyplot as plt

x = list(range(-5, 11))
y = [(i-3)**2 + 1 for i in x]

plt.plot(x, y, "--")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```

Dans les exemples précédents, on s'est servi du troisième argument de la fonction plot pour spécifier le style du tracé. On peut le faire varier, ainsi que spécifier des labels pour les différents tracés. Par exemple :

```
import matplotlib.pyplot as plt

x = list(range(-5, 11))
y1 = [(i-3)**2 + 1 for i in x]
y2 = [3*i for i in x]

plt.plot(x, y1, "r", label="quadratique")
plt.plot(x, y2, "b", label="linéaire")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend()
plt.show()
```

Enfin, il est possible de combiner plusieurs graphiques au sein d'une même figure, comme dans l'exemple suivant :

```
import matplotlib.pyplot as plt

# Données pour les graphiques
x = list(range(1,200,10))
y_lin = [2*i for i in x]
y_quad = [i**2 for i in x]  # Fonction quadratique

# Création de la figure et des axes (# 1 ligne, 2 colonnes)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Premier plot
```

```python
ax1.plot(x, y_lin, 'r-')  # 'r-' pour une ligne rouge
ax1.set_title('Fonction Linéaire')
ax1.set_xlabel('x')
ax1.set_ylabel('y')

# Deuxième plot
ax2.plot(x, y_quad, 'b-')  # 'b-' pour une ligne bleue
ax2.set_title('Fonction Quadratique')
ax2.set_xlabel('x')
ax2.set_ylabel('y')

# Affichage des graphiques
plt.tight_layout()  # Ajuste automatiquement les sous-graphiques
plt.show()
```

La librairie propose de nombreuses autres options, notamment pour créer des histogrammes ou visualiser des champs de valeurs, que le lecteur peut aisément explorer en consultant la documentation officielle de Matplotlib ou les ressources disponibles sur Internet.

**Exercice 9.8 - Création de graphiques simples**

Créez deux graphiques simples en utilisant Matplotlib. Le premier doit tracer une fonction linéaire $y = 2x + 1$ pour $x$ allant de $-10$ à 10. Le second doit tracer une fonction exponentielle $y = e^x$ sur le même intervalle. Utilisez des couleurs et des styles de lignes différents pour chaque graphique.

**Exercice 9.9 - Graphiques côte à côte**

Utilisez Matplotlib pour créer une figure contenant deux graphiques côte à côte. Le premier graphique doit afficher une parabole $y = x^2$ et le second une fonction cubique $y = x^3$. Assurez-vous d'ajouter des légendes et des titres appropriés pour chaque sous-graphique.

**Exercice 9.10 - Visualisation d'histogrammes**

Générez un ensemble de données normalement distribuées avec une moyenne de 4 et un écart-type de 1, et tracez un histogramme de ces données. Configurez les bins de l'histogramme pour qu'ils reflètent la distribution des valeurs. Ajoutez un titre et des étiquettes aux axes.

**Exercice 9.11 - Tir parabolique**

En mécanique classique, la trajectoire d'un projectile sans fottement lancé avec une vitesse initiale $v_0$ à un angle $\theta$ par rapport à l'horizontale est donnée par les équations suivantes :

$$x(t) = v_0 \cos(\theta)t$$

$$y(t) = v_0 \sin(\theta)t - \frac{1}{2}gt^2$$

Ecrivez un code qui demande les paramètres initiaux à l'utilisateur puis affiche le graphique de la trajectoire du projectile.

## 9.6 Écriture et lecture de fichiers

### 9.6.1 Lecture et écriture de fichiers texte

Lorsqu'on analyse des données en programmation scientifique, il est extrêmement commun de devoir ouvrir des fichiers pour y récupérer des données, ou bien pour y écrire des résultats. Bien qu'il existe des librairies très répandues pour manipuler les fichiers de données ainsi que les données elles-même (notamment, la très célèbre librairie *Panda*), nous examinons ici comment le faire nous-même à partir des fonctions de bases permises par Python.

En Python, il est possible d'ouvrir un fichier en mode lecture ou en mode écriture, en utilisant la fonction `open`. Par exemple, pour ouvrir un fichier en mode lecture :

```
fichier = open("mon_fichier.txt", "r") #ouvre le fichier en mode lecture
lines = fichier.readlines() #récupère les lignes du fichier
fichier.close() #ferme le fichier
```

Cependant, nous déconseillons fortement d'utiliser cette méthode, car elle ne ferme pas le fichier automatiquement, ce qui peut mener à des problèmes si le programme se termine par erreur avant que le fichier ne soit fermé. Il est préférable d'utiliser la syntaxe suivante, qui garantit que le fichier sera fermé automatiquement :

```
with open("mon_fichier.txt", "r") as fichier:
    lines = fichier.readlines() #récupère les lignes du fichier
```

Comme dit précédemment, il existe de nombreuses librairies pour manipuler automatiquement des données au format .csv, .xlsx, .json, etc. Nous allons nous contenter ici de montrer comment lire et écrire des fichiers textes simples, car cela illustre les fondements de la manipulation de fichier ; ainsi, même dans les cas malheureux où les données ne sont pas correctement formatées, il nous sera possible de construire un traitement automatique de leur contenu.

```python
with open("mon_fichier.dat", "r") as fichier:
    lines = fichier.readlines() #on récupère les lignes
    for line in lines:
        print(line) #on affiche chaque ligne
```

Par ailleurs, l'écriture au sein d'un fichier se fait comme suit, sans oublier de convertir en chaîne de caractères toute valeur à écrire dans le fichier :

```python
mes_valeurs = [1, 2, 3, 4, 5]
with open("mon_fichier.dat", "w") as fichier:
    for valeur in mes_valeurs:
        fichier.write(str(valeur) + "\n") #\n = retour à la ligne
```

### 9.6.2 Traitement des exceptions

La plupart des langages de programmation offre des mécanismes qui permettent de gérer les erreurs pouvant survenir lors de l'exécution d'un programme. En Python, cela se fait à l'aide des blocs `try`, `except` et `finally` – la gestion de ces mécanismes sort du cadre de ce cours ; cependant, dans le cas précis de la manipulation de fichiers, nous donnons un aperçu de la démarche à effectuer. Voici un exemple d'utilisation de ces blocs pour gérer les erreurs lors de l'ouverture d'un fichier :

```python
try:
    with open("mon_fichier.txt", "r") as fichier:
        lines = fichier.readlines()  # récupère les lignes du fichier
    print("Lecture réussie !")
    print(lines)
except FileNotFoundError:
    print("Erreur : le fichier spécifié n'a pas été trouvé.")
except IOError:
    print("Erreur d'entrée/sortie lors de l'accès au fichier.")
except Exception as e:
    print("Une autre erreur est survenue :", e)
```

### 9.6.3 Écriture et lecture de structures de données avec Pickle

Pour terminer, notons que le module `pickle`, qui vient nativement avec Python, permet de sauvegarder des objets Python dans des fichiers, et de les recharger par la suite. Cela peut être très utile pour sauvegarder des structures de données standard mais complexes, comme des listes de listes, des dictionnaires, ou des objets de classes personnalisées. Voici un exemple simple d'utilisation de `pickle` :

```python
import pickle
ma_liste = [1, 2, 3, 4, 5]
mon_dict = {"a":1, "b":2, "c":3}
with open("mon_fichier.pkl", "wb") as fichier:
    pickle.dump(ma_liste, fichier)
    pickle.dump(mon_dict, fichier)
```

Cela a pour effet d'écrire automatiquement les objets `ma_liste` et `mon_dict` dans le fichier binaire `mon_fichier.pkl`. Pour les recharger, il suffit de faire :

```python
with open("mon_fichier.pkl", "rb") as fichier:
    ma_liste = pickle.load(fichier)
    mon_dict = pickle.load(fichier)
```

### Exercice 9.12 - Tracé de données réelles

Chargez un ensemble de données depuis un fichier CSV contenant deux colonnes, $x$ et $y$, et tracez les données dans un graphique à points (scatter plot). Assurez-vous d'ajouter des étiquettes aux axes et un titre au graphique. Utilisez des croix rouges pour symboliser les points.

### Exercice 9.13 - Lecture de fichier et traitement des données

Écrivez un script Python qui ouvre un fichier texte nommé `"data.txt"` contenant plusieurs lignes de données numériques, chaque ligne étant séparée par des virgules. Le script doit lire les données, calculer la moyenne de chaque ligne et imprimer ces moyennes.

**Exemple d'output :**

```
Moyenne ligne 1 : 5.2
Moyenne ligne 2 : 7.3
...
```

### Exercice 9.14 - Gestion des erreurs de fichier

Modifiez l'exercice précédent de lecture de fichier pour ajouter la gestion des exceptions. Le script doit gérer les cas où le fichier n'existe pas, où il ne peut pas être lu, ou contient des données non numériques, en affichant un message d'erreur approprié sans interrompre brutalement l'exécution.

**Exemple d'output :**

```
Erreur : Le fichier spécifié n'existe pas.
ou
Erreur : Données non numériques trouvées.
```

### Exercice 9.15 - Écriture dans un fichier avec formatage

Créez un script Python qui génère un fichier `"output.txt"`. Le fichier doit contenir les carrés des nombres de 1 à 10, chaque nombre et son carré étant sur une nouvelle ligne, séparés par une tabulation.

**Exemple d'output :**

```
1	1
2	4
...
10	100
```

# Deuxième partie

# Algorithmique

# Chapitre 10

# Introduction à l'algorithmique

## 10.1 Qu'est-ce qu'un algorithme ?

Un algorithme est une méthode de résolution systématique d'un problème. Souvent, on présente les algorithmes comme des « recettes » mathématiques. Cette comparaison est pertinente car une recette présente un caractère systématique : elle ne varie pas et permet invariablement de réaliser un plat donné, pourvu que toutes les conditions soient respectées à la lettre.

Afin qu'un algorithme puisse être utilisable, il est indispensable que les différentes étapes qui le constituent soient non ambigues. Évidemment, il faut également que le problème soit suffisamment bien défini. Dans l'idée, personne ne peut proposer de recette pour « faire un dessert » ; en revanche, on peut proposer une recette pour « faire un gâteau au chocolat pour 6 personnes ».

Lors du demi-siècle passé, un grand effort a été fourni pour développer des algorithmes dans une myriades de domaines. En effet, l'avènement de l'ordinateur moderne a permis d'appliquer des méthodes de résolution de problèmes de plus en plus complexes ; inversément, pour pouvoir exploiter au mieux la puissance grandissante des machines, l'invention de nouveaux algorithmes s'est avérée nécessaire. Même si, de nos jours, l'utilisation d'ordinateurs pour *implémenter* des algorithmes est courante, il faut garder en tête qu'un algorithme n'est pas un code informatique : c'est une « recette mathématique » abstraite qui peut exister en-dehors de toute machine.

## 10.2 Un premier exemple

Afin de donner un exemple différente de celui d'une recette de cuisine, nous allons ici proposer un algorithme pour dessiner une spirale à angles droits sur du papier quadrillé, qui tourne dans le sens horaire, et dont le plus grand côté est aussi long que 5 carrés. Au préalable, on suppose donc que l'utilisateur de l'algorithme dispose d'une feuille quadrillée suffisamment grand et d'un crayon.

```
--- début ---

1) Positionner la pointe du crayon au croisement le plus en haut et
le plus à gauche de la page.

2) Depuis la position actuelle du crayon, traçer une droite d'une
longueur de 5 carrés le long du quadrillage, vers la droite.

3) Depuis la position actuelle du crayon, traçer une droite d'une
longueur de 4 carrés le long du quadrillage, vers le bas.

4) Depuis la position actuelle du crayon, traçer une droite d'une
longueur de 3 carrés le long du quadrillage, vers la gauche.

5) Depuis la position actuelle du crayon, traçer une droite d'une
longueur de 2 carrés le long du quadrillage, vers le haut.

6) Depuis la position actuelle du crayon, traçer une droite d'une
longueur de 1 carrés le long du quadrillage, vers la droite.

--- fin ---
```

Cet algorithme fonctionne. Cependant, on peut lui trouver deux défauts majeurs : premièrement, il est quelque peu **répétitif**, et deuxièmement, il n'est pas **général**. Si maintenant nous voulons un algorithme pour une spirale dont le côté le plus long est de cent carrés, nous devons réécrire une autre méthode, qui sera d'ailleurs bien longue.

Voici donc une **généralisation** de l'algorithme précédent, qui permet de dessiner une spirale dont le plus long côté est de $N$ carrés :

```
--- début ---

1) Positionner la pointe du crayon au croisement le plus en haut et
le plus à gauche de la page.

2) La direction actuelle du trait est vers la droite.

3) Tracer une droite de N carrés le long du quadrillage.
La droite se dirige dans la direction actuelle du trait.

4) Tourner la direction actuelle du trait de 90° dans le sens horaire.

5) Soustraire 1 à la valeur actuelle de N.

6) Si N vaut zéro, arrêter l'algorithme. Sinon, retourner à l'étape 3).

--- fin ---
```

L'avantage de ce nouvel algorithme et qu'on peut s'en servir pour tracer une spirale à 1000 côtés ou à 5 côtés sans rien changer ! Il faut également noter que cet algorithme prend un **paramètre d'entrée** : $N$. Les paramètres d'entrée sont des variables du problème que l'on cherche à résoudre.

Dans ce cours, nous associerons souvent des schémas, que nous appellerons **algorigrammes**, aux algorithmes que nous étudierons. La figure 10.1 ci-dessous montre l'algorigramme associé à notre algorithme général de la spirale :

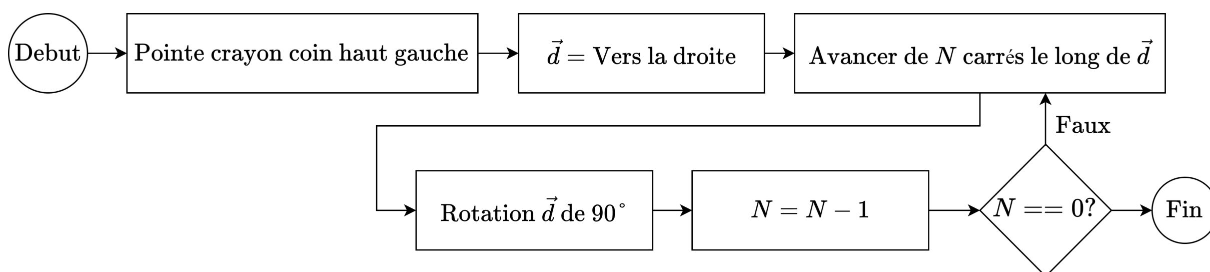



FIGURE 10.1 – Algorigramme du dessin d'une spirale sur une feuille quadrillée. La rotation s'entend dans le sens horaire. On discerne aisément la boucle qui permet de répéter les étapes 3 à 5.

Dans les algorigrammes de ce cours, les instructions sont encadrées, les conditions sont écrites dans un losange et le début et la fin sont dans des cercles. Le début et la ou les fins sont écrites dans un cercle. Les flèches indiquent le sens de la logique, et son nécessaires dans la plupart des cas pour éviter de rendre l'interpretation du diagramme ambigue. Enfin, on indique au moins l'une des valeurs des branches qui sortent d'une condition.

## 10.3 Exercices

**Exercice 10.0**

Implémentez l'algorithme de la spirale avec le module turtle de Python.
Vous devez créer une fonction nommée `draw_spiral` qui prend $N$ et $k$ en entrée. $N$ est tel que décrit plus haut, tandis que $k$ est le nombre de carrés (pixels) que l'on soustrait à chaque étape. Dans l'exemple du haut, $k = 1$.

**Exercice 10.1**

Adaptez la fonction de l'exercice précédent pour passer un troisième paramètre, $a$, qui est le nombre de degrés dont $d$ tourne à chaque étape.

## Exercice 10.2

Proposez un algorithme permettant de tracer un polygone régulier et qui prend $N$ et $c$ en paramètres. $N$ est le nombre de côtés du polygone, $c$ est la longueur des côtés.
A) Donnez l'algorithme sous forme d'algorigramme.
B) Finalement, programmez une fonction `polygon` qui implémente cet algorithme en utilisant turtle.

## Exercice 10.3 — Min et Max

A) Proposez un algorithme sous forme d'algorigramme afin de trouver la valeur minimale au sein d'une liste de nombres.
B) Implémentez votre algorithme en Python. Testez le sur la liste suivante :

```
liste1 = [3.14, -6.33, 10023, -0.0001, 0.55, 0.00001, 44]
```

C) Pour tester votre algorithme davantage, comparez la réponse qu'il fournit à celle de la fonction `min` de Python, qui s'utilise comme suit :

```
liste1 = [3.14, -6.33, 10023, -0.0001, 0.55, 0.00001, 44]
min_selon_python = min(liste1)
print(min_selon_python) #affiche -6.33
```

D) Pour tester votre algorithme davantage, comparez la réponse qu'il fournit à celle de la fonction `min` de Python sur 1000 listes générées aléatoirement, chacune d'une longueur de 1000 nombres. Voici un exemple de génération de liste d'entiers aléatoires :

```
import random #importe le module pour générer des nb aléatoires
val_min = -1000000 #valeur min des nombres aléatoires
val_max = 1000000 #valeur max des nombres aléatoires
N = 100 #longueur de la liste aléatoire
liste_rand = [random.randint(val_min, val_max) for i in range(N)]
```

## Exercice 10.4

Donnez l'algorigramme des exercices 7.13, 7.15 et 7.16.

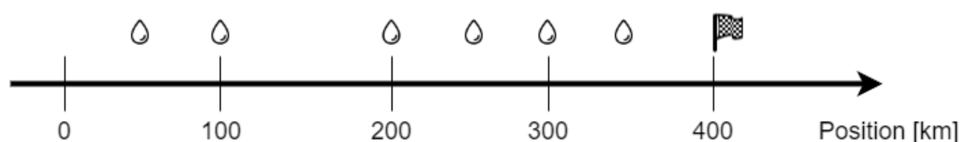
# Chapitre 11

# Stratégies gloutonnes

Les algorithmes gloutons désignent une stratégie de résolution de problème. Résoudre un problème de façon gloutonne, c'est construire une solution sans jamais « revenir en arrière » pour changer d'avis. En général, on a affaire à des problèmes où il faut choisir, à chaque instant du problème, une solution provisoire qui semble être la meilleure. Toute la difficulté réside alors dans le fait de trouver ce que veut dire « meilleur ».

## 11.1 Exemple de la traversée du désert

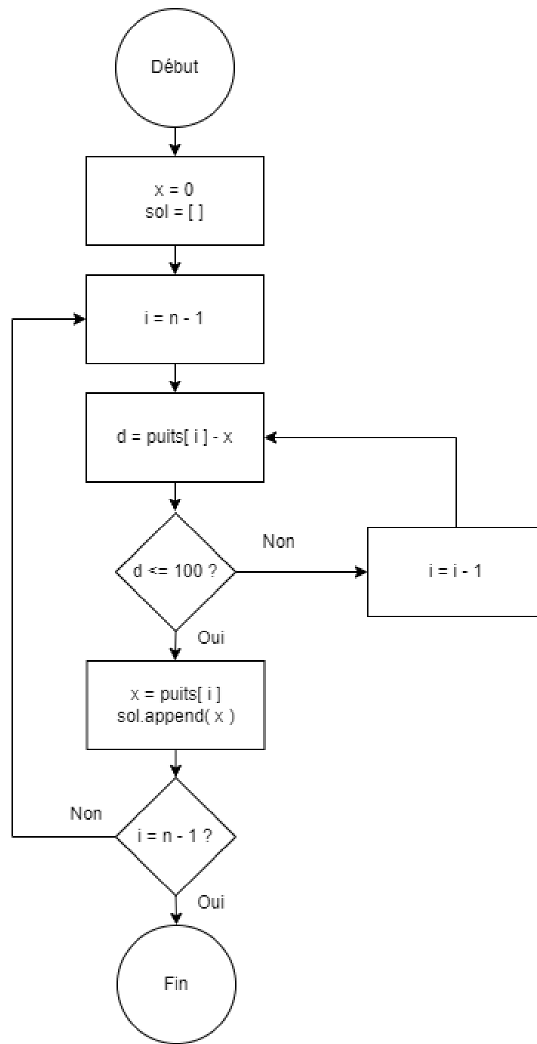
Dans cet exemple, un voyageur cherche à traverser un désert. Le désert fait 400 km de long. Le voyageur ne peut marcher plus de 100 km sans s'hydrater. Pour s'hydrater, il a besoin de puits, qui sont disposés aux endroits suivants (on ne précise plus qu'il s'agit de kilomètres) : 50, 100, 200, 250, 300, 350. À quels puits le voyageur doit-il s'arrêter s'il veut effectuer le moins d'arrêts possible ?




La stratégie gloutonne, dans ce problème, consiste à toujours choisir le puits atteignable (sans mourir de soif) le plus loin possible de la position actuelle du voyageur.

En appliquant cette stratégie, on obtient la solution suivante : les puits auxquels il faut s'arrêter se trouvent à 100, 200 et 300 km. Nous n'allons pas ici prouver mathématiquement que cette stratégie permet forcément de faire le moins d'arrêts possibles. En revanche, on peut facilement trouver des exemples pour lesquels la stratégie ne donne pas l'une des autres solutions possibles.

On peut donner l'algorigramme de cet algorithme. Dans l'algorigramme qui suit, on nomme `puits` la liste des puits (l'élément numéro $i$ contient la position du puits numéro $i$), et `n` désigne le nombre de puits.




Voici le code Python qui correspond à ce même algorithme :

```
puits = [50, 100, 150, 200, 250, 350]
n = len(puits)

x = 0
sol = []

while True:
    i = n - 1
    while puits[i] - x > 100:
        i = i - 1
    x = puits[i]
    sol.append(x)
    if x == puits[n-1]:
        break

print(sol)
```

## 11.2 Exercices

**Exercice 11.0 — Traversée du désert alternative**

Trouvez un exemple de disposition de puits, dans le problème du désert, où il existe des solutions aussi courtes que celle donnée par l'algorithme glouton vu dans le cours.

**Exercice 11.1 — Traversée du désert alternative (2)**

Implémentez l'algorithme de traversée du désert en Python, et cherchez la solution pour le cas où le désert fait 1000 km de long et où les puits sont disposés comme suit :

```
puits = [50, 200, 300, 350, 450, 550, 600, 650, 700, 850, 900, 1000]
```

Par ailleurs, on peut maintenant effectuer des étapes de 150 km sans mourir de soif ! Quelle est la solution donnée par l'algorithme ?

**Exercice 11.2 — Sac à dos**

Votre sacoche possède un volume de $1,5$ litres et vous voulez que les objets qu'elle contient représentent la plus grand somme d'argent possible. Voici les objets disponibles, avec leur volume et leur prix. Vous ne pouvez pas prendre plusieurs fois le même objet.

```
objets = ["ordinateur", "diamant", "bouteille d'eau", "biscuit",
            "smartphone", "masque FFP2", "Bâton", "Sac de blé",
            "pierre semi-précieuse"]
volumes = [1, 0.001, 1, 0.1, 0.2, 0.05, 1.2, 10, 0.1] #en litres
prix = [450, 5000, 1, 1, 300, 15, 0.5, 20, 50] #en CHF
```

Une solution possible, par exemple, est de prendre le lingot d'or et la bouteille d'eau. Cette solution fonctionne car le volume totale vaut 1.5 litre. La valeur totale du sac est alors de 10'001 CHF.
**But :** écrivez un algorithme qui permette de sélectionner les meilleurs objets à mettre dans le sac pour que la valeur totale de ce dernier soit maximale.

**Exercice 11.3 — Pot de peinture**

Téléchargez l'image dessin.png ainsi que le code coloriage.py. Adaptez le code coloriage.py pour colorier une zone fermée du dessin avec la couleur rouge à partir du pixel cliqué par l'utilisateur (fonctionnalité "pot de peinture").

**Exercice 11.4 — Rendu de monnaie britannique**

Dans le temps, la monnaie britannique contenait les pièces suivantes (notez que « pence » est le pluriel de « penny ») :

- la *demi-couronne* d'une valeur de 30 pence ;
- le *florin* d'une valeur de 24 pence ;
- le *shilling* d'une valeur de 12 pence ;
- le *sixpence* d'une valeur de 6 pence ;
- le *threepence* d'une valeur de 3 pence ;
- le *penny*.

1. Donnez un pseudocode correspondant à l'algorithme Greedy *vu en cours* pour résoudre ce problème.
2. Implémentez cet algorithme en Python. Testez-le sur la monnaie à rendre sur 100 pence. L'algorithme doit donner le nombre de pièces de chaque sorte que l'on rend.

```
total = 100
liste_pieces = [30, 24, 12, 6, 3, 1]
solution = []
... #ici, on code l'algorithme
print(solution)
```

**Exercice 11.5 — Trier une liste**

Proposez un algorithme capable de trier une liste de n'importe quelle longueur de façon croissante. La liste contient des nombres. Par exemple :

```
liste = [3,9,34,-12,1]
# ==> après tri, la liste doit valoir [-12,1,3,9,34]
```

Testez votre algorithme en l'implémentant en Python.

## 11.3 Tri de liste

Le tri de liste est un problème qui peut être abordé de plusieurs façon différentes. Une façon intuitive d'aborder le problème est de le formuler de façon « gloutonne », en cherchant à créer une nouvelle liste $b$ à partir d'une liste $a$, en prenant à chaque fois le plus petit nombre restant au sein de $a$ pour le déplacer dans $b$. La figure 11.1 montre un algorigramme possible de cet algorithme.

Quelle est l'efficacité de cet algorithme ? On peut calculer le nombre d'étapes nécessaires, si la longueur de la liste à trier est $n$. Tout d'abord, on va parcourir chaque élément de la liste une fois. À chaque fois, on devra trouver le minimum restant dans la liste, ce qui

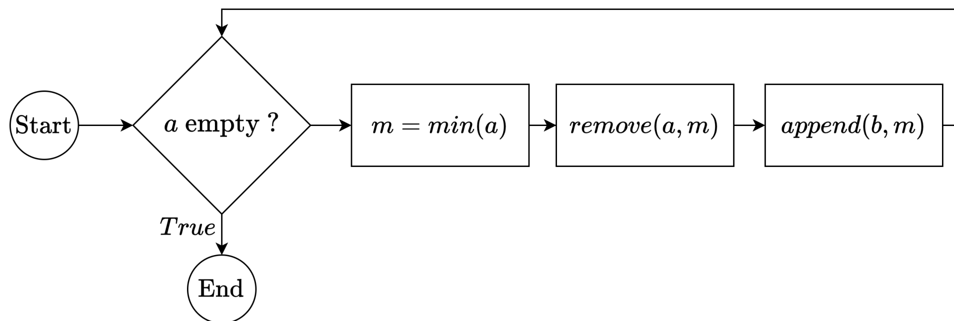



FIGURE 11.1 – Algorigramme du tri naïf, où l'on suppose que les listes `a` et `b` existent, ainsi que les fonction pour trouver le minimum, pour supprimer un élément et pour ajouter un élément à une liste.

nécessite de parcourir la liste entière. Ensuite, on devra supprimer l'élément de la liste, ce qui nécessite de parcourir la liste entière. Enfin, on devra ajouter l'élément à la liste $b$, ce qui ne nécessite pas de parcourir cette dernière. Le nombre $N$ d'étapes nécessaires vaut donc environ :

$$N \approx \sum_{i=1}^{n} i + i = n(n-1) \approx n^2, \tag{11.1}$$

où l'on a négligé les opérations de comparaison au sein de la recherche du minimum, entre autres.

Notre objectif, à présent, est de trouver un algorithme plus performant que ce dernier.

### 11.3.1 Tri par sélection

Il est possible de formuler une version du tri naïf qui ne requiert pas de créer une nouvelle liste tout en ôtant les minimums successifs de la liste de base. Cette version permet de faire un tri **en place**, c'est-à-dire que l'on ne crée pas de nouvelle liste, mais que l'on modifie directement la liste de base. L'idée de l'algorithme se résume ainsi :

1. Tout en parcourant la liste à partir de l'indice $i_{\text{début}}$, on cherche l'indice $i_{\text{min}}$ du plus petit élément.
2. Une fois la liste parcourue, ou échange l'élément minimum avec l'élément à l'indice $i_{\text{début}}$.
3. On incrémente $i_{\text{début}}$ et on recommence le processus depuis le point 1.

Evidemment, on choisira $i_{\text{début}} = 0$ au commencement de l'algorithme. Le code Python suivant montre une façon d'implémenter cet algorithme :

```
1  def tri_selection(a):
2      n = len(a)
3      for i_debut in range(n):
4          i_min = i_debut
5          for i in range(i_debut+1, n):
6              if a[i_min] > a[i]:
7                  i_min = i
```

```
        a[i_debut], a[i_min] = a[i_min], a[i_debut] #swap de fin de parcours
    return a
```

Le nombre d'étapes significatives de cet algorithme est comme suit : au premier parcours, on doit nécessairement effectuer $n$ comparaisons, puis $n-1$ au second parcours, et ainsi de suite durant $n$ parcours distincts. Le nombre d'étapes est donc environ :

$$N_{\mathrm{sel}} = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}. \tag{11.2}$$

### 11.3.2 Tri à bulles

Examinons un autre algorithme de tri inspiré de la méthode gloutonne. L'idée de l'algorithme est de comparer les éléments de la liste deux à deux, et de les échanger (swap en anglais) si nécessaire, sans jamais comparer chaque éléments avec tous les autres.

Le point central, ici, est de voir qu'il suffit de s'en tenir aux comparaisons entre nombres voisins au sein de la liste. Si l'on « remonte » la liste en échangeant chaque pair de voisins qui ne respecte pas un ordre croissant, on peut être sûr que le plus grand nombre de la liste se retrouvera à la fin. Or, l'opération peut être répétée en ne considérant cette fois qu'une liste plus courte (à l'image de ce qui a été fait pour le tri sélection), car on peut s'arrêter avant le dernier élément, qui est forcément au bon endroit. On peut alors répéter l'opération jusqu'à ce que la liste soit entièrement triée.

Le code Python suivant montre comment implémenter cet algorithme :

```
def tri_bulle(a):
    n = len(a)
    for i_debut in range(n):
        i_fin = n - i_debut - 1
        for i in range(0, i_fin): #premiere iteration : i_fin = n - 1
            if a[i] > a[i+1] :
                a[i], a[i+1] = a[i+1], a[i]
    return a
```

Combien d'étapes significatives sont ici à considérer ? À la première itération, on doit comparer $n-1$ éléments. À la deuxième itération, on doit comparer $n-2$ éléments, et ainsi de suite. Le nombre total d'étapes est donc le même que pour le tri par sélection, soit environ :

$$N = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}, \tag{11.3}$$

ce qui est approximativement moitié moins que le tri naïf.

Le tri à bulles peut aisément être amélioré en comptant le nombre d'échanges effectués à chaque parcours de liste ; si aucun échange n'est effectué, la liste est déjà triée et

l'algorithme peut s'arrêter en avance.

Comme nous le verrons dans le prochain chapitre, le fait que nos trois méthodes gloutonnes prennent de l'ordre de $n^2$ étapes pour trier $n$ éléments nous pousse à les considérer, peu ou prou, comme équivalentes du point de vue de leur efficacité. En effet, il existe des méthodes de tri plus efficaces, qui prennent de l'ordre de $n \log(n)$ étapes pour trier $n$ éléments. Nous verrons ces méthodes dans le prochain chapitre.

# Chapitre 12

# Diviser pour régner

Les algorithmes gloutons, vus dans le chapitre précédent, sont souvent simples à mettre en place. En revanche, dans de nombreux cas il existe des algorithmes plus pertinents pour des problèmes simples. L'une de ces classes d'algorithmes se nomme « diviser pour régner » (divide and conquer en anglais).

## 12.1 Principe

Le principe des algorithmes du type diviser pour régner est de décomposer un problème en sous-problèmes plus petits, de résoudre ces sous-problèmes, puis de combiner les solutions pour obtenir la solution du problème initial.

### 12.1.1 Recherche dichotomique

Prenons un exemple concret : on dispose d'une liste de nombres triés par ordre croissant et on cherche l'emplacement d'un nombre bien précis au sein de cette liste. On peut alors appliquer la démarche suivante (volontairement peu détaillée) :

1. On compare le nombre cherché avec le nombre au milieu de la liste.
2. Si le nombre cherché est plus petit que le nombre au milieu de la liste, on revient au point 1 après avoir décidé que la nouvelle liste à considérer est la première moitié de la liste initiale.
3. Si le nombre cherché est plus grand que le nombre au milieu de la liste, on revient au point 1 après avoir décidé que la nouvelle liste à considérer est la deuxième moitié de la liste initiale.
4. Si le nombre cherché est égal au nombre au milieu de la liste, on a trouvé l'emplacement du nombre cherché.

La figure 12.1 montre l'algorigramme de cet algorithme, encore incomplet pour des raisons que nous expliquons plus loin. Tout d'abord, dans le cas où la liste est de longueur $L$ impaire, il faut trancher pour savoir si l'on coupe trop à gauche ou trop à droite. Ici, on arrondit donc vers le bas l'indice $i$ qui détermine l'endroit où l'on coupe. Par ailleurs,

comme on le voit dans l'algorigramme, tout un bloc d'instructions se répète. Ce bloc est encadré en traitillés et nous lui donnons le nom de `cherche(k,a)`, où $k$ est le nombre cherché et $a$ est la liste dans laquelle on cherche le nombre.

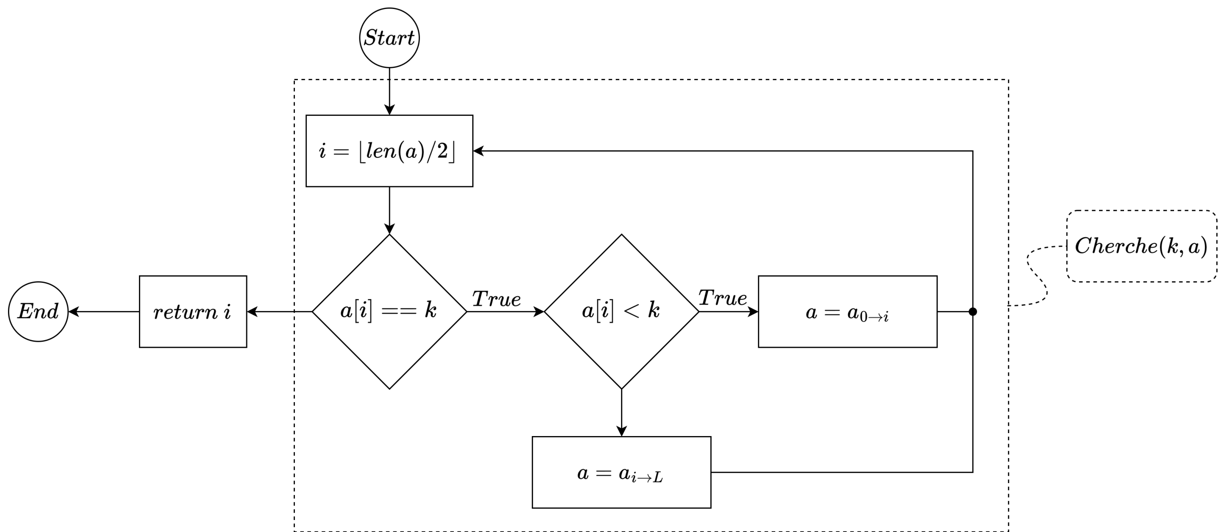



FIGURE 12.1 – Algorigramme de la recherche dichotomique **presque** correct.

Il ne nous reste plus qu'à écrire un code qui réalise cet algorithme. Nous avons cependant un dernier détail à régler : comme on veut connaître l'emplacement du nombre cherché, on renvoie l'indice de ce nombre dans la liste. Or, dans le cas où l'on doit chercher dans la seconde moitié de la liste, il faut ajouter l'indice de début de cette moitié à l'indice trouvé dans la seconde moitié. En d'autres termes, il ne faut pas perdre la trace de l'indice de début de la liste initiale lorsqu'on entre dans une nouvelle récursion de l'algorithme.

Le code ci-dessous implémente la recherche dichotomique en Python :

```python
def cherche(k,a):
    i = int(len(a) / 2) #int arrondit vers le bas si argument non entier
    if a[i] == k:
        return i
    else:
        if a[i] < k:
            return i + cherche(k,a[i:]) #ne pas oublier le décalage !
        else:
            return cherche(k,a[:i])

a = [1, 4, 7, 8, 9, 10, 14, 16] # liste triée
print(cherche(8,a)) # ceci affiche bien la valeur 3
print(cherche(4,a)) # ceci affiche bien la valeur 1
```

Il est important de remarquer que la fonction `cherche` s'appelle elle-même (sur de nouveaux arguments). C'est ce que l'on nomme une **fonction récursive**. Grâce à cela, nous avons créé un algorithme de recherche sans avoir eu à utiliser de boucle.

Une fonction récursive, pour se terminer, doit toujours au moins comprendre un cas de récursion (ici en ligne 5) et un cas de base (ici en ligne 3). Le cas de base est celui où l'on sait immédiatement quelle est la solution du problème, souvent parce que le problème a été suffisamment découpé pour devenir trivial. Dans notre cas, c'est lorsque l'on trouve le nombre cherché dans la liste.

### 12.1.2 Comparaison avec un algorithme glouton

Un algorithme glouton (ou naïf) qui résoudrait le problème de la recherche d'un élément au sein d'une liste triée consisterait à parcourir la liste de gauche à droite jusqu'à trouver le nombre cherché. Nous écrivons directement son code en Python :

```python
def cherche_glouton(k,a):
    for i in range(len(a)):
        if a[i] == k:
            return i
```

C'est un algorithme qui a l'avantage d'être particulièrement simple à comprendre et à implementer. En quoi, alors, l'algorithme naïf est-il plus mauvais que l'algorithme diviser pour régner ? Pour répondre à cette question, examinons le nombre d'étapes à effectuer afin d'arriver au résultat, dans les deux cas.

Dans le meilleur des cas, l'algorithme naïf trouve l'élément dès la première itération. Ainsi, il effectue une seule étape. Dans le pire des cas, l'élément cherché est le dernier de la liste, et l'algorithme effectue $n$ étapes, où $n$ est la longueur de la liste. De façon générale, si l'élément à chercher se trouve en $i$-ème position, il faut $i$ étapes pour le trouver. En moyenne, le nombre d'étapes que doit effectuer l'algorithme naïf vaut donc :

$$N_{\text{naif}} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n-1)}{2} = \frac{n-1}{2}. \tag{12.1}$$

À présent, examinons le nombre d'étapes que doit effectuer l'algorithme diviser pour régner. Dans le pire des cas, l'algorithme effectue également une seule étape (cas où l'élément à chercher est déjà au milieu de la liste). Le pire des cas, lui, est plus avantageux : en effet, le nombre d'étapes à effectuer correspond au nombre de fois où il faut diviser la longueur de la liste par 2 avant d'arriver à une liste d'un seul élément. Le pire des cas vaut donc $\log_2(n)$. Ainsi, on peut écrire (sans le prouver [1]) que :

$$N_{\text{div}} = \log_2(n), \tag{12.2}$$

et on comprend que l'algorithme diviser pour régner est bien plus efficace en général, et ce d'autant plus que $n$ est grand !

1. Mais ce n'est pas grave dans l'optique de comparer les deux algorithmes, puisque nous adoptons une démarche conservative, où nous prenons le cas moyen pour le pire des cas.

### 12.1.3 Complexité et performances

Pour comparer deux algorithmes, on peut s'intéresser à leur complexité. La complexité d'un algorithme est une mesure de l'effort nécessaire pour exécuter cet algorithme. On peut distinguer plusieurs types de complexité, mais nous nous intéressons ici à la **complexité en temps**, qui est une mesure du nombre d'opérations nécessaires pour exécuter l'algorithme.

Plus haut, nous avons comparé le « nombre d'étapes » pris par l'un ou l'autre de deux algorithmes, sans davantage préciser ce que nous entendions par là. Il est important de comprendre que ce qui domine le coût en temps d'un algorithme informatique est, sauf cas extrêmes où l'on manipule un très petit problème, le nombre d'opérations élémentaires à effectuer. Il faut se méfier de cette vision des choses cependant, car une division peut être bien plus coûteuse qu'une addition, par exemple, mais une discussion plus complète du sujet sort du cadre de ce cours.

Dans l'algorithme naïf de la recherche dichotomique, l'opération élémentaire est la comparaison entre deux nombres (ligne 3 du code). On dit alors que la complexité en temps de cet algorithme est en $\mathcal{O}(n)$, où $n$ est la longueur de la liste. Cela doit se comprendre comme « la complexité en temps de l'algorithme est linéaire en fonction de la longueur de la liste ».

En revanche, l'algorithme diviser pour régner effectue en moyenne $\log_2(n)$ comparaisons, ce qui donne une complexité en $\mathcal{O}(\log(n))$. Cela doit se comprendre comme « la complexité en temps de l'algorithme est linéaire en fonction de la longueur de la liste ». Cependant, on peut argumenter que contrairement à l'algorithme naïf, l'algorithme diviser pour régner effectue non seulement une comparaison (ligne 3, cas de base), mais même une deuxième dans la plupart des cas (ligne 6, cas de récursion). Par ailleurs, une division est effectuée dans chaque cas (ligne 2) et une addition est même effectuée (ligne 7) dans la moitié des cas en moyenne [2]. On pourrait donc écrire qu'en prenant en compte ces opérations, la complexité en temps de l'algorithme diviser pour régner est en $\mathcal{O}(c\log(n))$, où $c$ est une constante associée au coût des opérations supplémentaires par rapport à l'algorithme naïf. Bien que ce genre de détails puissent avoir de l'importance en pratique, cela ne change pas la *complexité* de l'algorithme, puisque le temps mis par ce dernier pour résoudre un problème de taille $n$ est bien proportionnel à $\log(n)$. En particulier, on peut ignorer les constantes parce que :

- Dans les cas où $n$ est grand, le comportement de la fonction (logarithmique, linéaire, polynomial d'un certain degré, exponentiel, etc.) est bien plus important que la constante pour refléter le temps nécessaire pour résoudre le problème. Par exemple, il existera toujours une valeur finie de $n$ à partir de laquelle un algorithme de complexité logarithmique sera moins long qu'un algorithme de complexité linéaire, quelle que soit la valeur (finie) des constantes multiplicatives de chacun !
- Les constantes dépendent fortement de l'implémentation de l'algorithme, du langage

2. Quant au coût de communiquer la nouvelle sous-liste pour la prochaine récursion, il dépend fortement du langage et de la façon dont sont implémentées les listes et nous n'en parlons pas ici, bien que ce soit un sujet important. En l'occurrence, il faut prendre garde que la complexité en temps de l'opération « ignorée » ne soit pas pire que celle de l'algorithme global. Dans tous les cas, il est aisé ici de formuler une version équivalente de l'algorithme où seuls les indices de début et de fin de la recherche sont modifiés (au prix d'une soustraction supplémentaire), et où la liste elle-même n'est pas passée en argument.

de programmation, de l'ordinateur utilisé, etc. Il est donc difficile de les comparer entre deux algorithmes différents.

Ainsi, la notation $\mathcal{O}$ est une notation dite « asymptotique », qui s'intéresse au comportement de la fonction pour des valeurs de $n$ très grandes. Les constantes n'ont donc pas d'importance dans ce contexte. C'est également pour cela que la base des exponentielles et des logarithmes n'importe pas pour la notation $\mathcal{O}$. Les figures 12.2 et 12.3 illustrent le comportement de différentes fonctions en fonction de la taille de l'entrée. La table 12.1 donne quelques complexités courantes, avec des exemples d'algorithmes concernés.

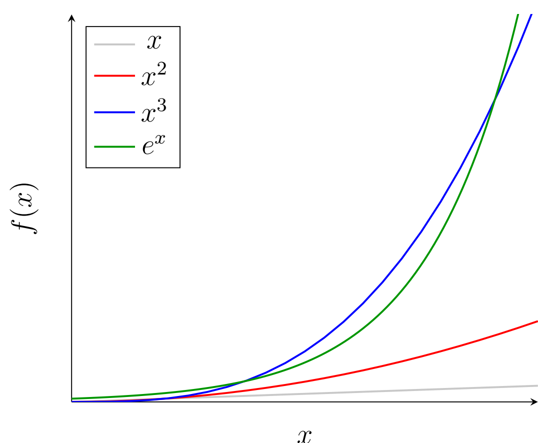



FIGURE 12.2 – Trois fonctions polynomiales et une fonction exponentielle. On voit que pour de petites quantités, la valeur est trompeuse ; la fonction exponentielle finit toujours par dépasser les fonctions polynomiales.

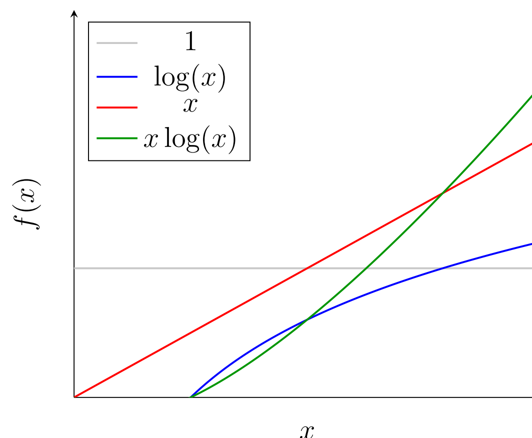



FIGURE 12.3 – Fonctions constante, logarithmique, linéaire et linéarithmiques, qui croissent toutes plus lentement que des polynômes de degré supérieur à 1.

TABLE 12.1 – Quelques types de complexité communs classés par croissance, avec des exemples d'algorithmes.

| Nom croissance | Complexité en temps | Exemples d'algorithmes |
|---|---|---|
| Constante | $\mathcal{O}(1)$ | Trouver si un nombre binaire est pair |
| Logarithmique | $\mathcal{O}(\log(n))$ | Recherche dichotomique |
| Linéaire | $\mathcal{O}(n)$ | Trouver le minimum d'une liste |
| Linéarithmique | $\mathcal{O}(n\log(n))$ | Tri fusion |
| Quadratique | $\mathcal{O}(n^2)$ | Calculer la couleur moyenne d'une image |
| Cubique | $\mathcal{O}(n^3)$ | Multiplication naïve de matrices $n \times n$ |
| Exponentielle | $\mathcal{O}(2^n)$ | Casser un mot de passe par force brute |
| Factorielle | $\mathcal{O}(n!)$ ou [3] $\mathcal{O}(n^n)$ | Générer toutes les permutations d'un ensemble |

Pour clore cette introduction à la complexité en temps, notons que l'on peut également s'intéresser à d'autres types de complexité, comme la complexité en espace (mémoire). Par ailleurs, il faut rester vigilant quant à la nature des opérations élémentaires que l'on prend en compte. Par exemple, considérons l'algorithme suivant :

```python
def tri_escroquerie(a): #ou plus honnetement, sleep_sort
    liste_triee = []
    for element in a:
        create_parallel_thread_and_sleep(element)
        liste_triee.append(element)
    return liste_triee
```

Dans cet algorithme, on crée un fil d'exécution parallèle (thread) pour chacun des nombres de la liste, puis l'on fait attendre (sleep) le thread d'une durée égale à la valeur du nombre. Quand le thread sort du sommeil, on ajoute le nombre qui lui est associé à la liste triée. Comme les threads issus de petits nombres sortiront du sommeil avant ceux issus de grands nombres, la liste triée sera bien la liste initiale. Or, le nombre d'étapes de cet algorithme n'est aucunement impacté par la taille de la liste ; sa complexité en temps est donc de $\mathcal{O}(1)$, ce qui est techniquement vrai pour l'algorithme, mais certainement pas pour son implémentation informatique, qui souffre de divers problèmes. Cet algorithme est inutilisable en pratique, d'une part pour le nombre de threads qu'il implique possiblement, d'autre part parce que pour des erreurs peuvent apparaître en raison du temps écoulé entre la création des différents threads, et enfin parce que le temps d'exécution dépend fortement de la valeur des nombres de la liste ! En fait, sa complexité en temps est $\mathcal{O}(M)$, où $M$ est la valeur maximale au sein de la liste (en supposant que l'on a affaire à des nombres positifs ; si tel n'est pas le cas, un prétraitement de la liste serait nécessaire, ce qui aurait une complexité $\mathcal{O}(n)$).

## 12.2 Tri fusion

Afin de mieux s'approprier la technique du diviser pour régner, intéressons-nous au problème du tri d'une liste non triée.

Le tri fusion est un algorithme de tri qui suit la méthode diviser pour régner. Il consiste à diviser la liste à trier en deux sous-listes de taille égale, trier ces sous-listes, puis fusionner les deux sous-listes triées pour obtenir la liste triée. L'algorithme est récursif, et s'arrête lorsque la liste à trier est de taille 1.

Commençons par nous intéresser à l'étape de fusion. Supposons que l'on dispose de deux listes triées $a_1$ et $a_2$. On souhaite les fusionner en une seule liste triée. Les cas de figures possibles sont les suivants :

- Cas trivial : si l'une des deux listes est vide, le résultat de la fusion est l'autre liste.
- Autres cas : si la première valeur de $a_1$ est inférieure à la première valeur de $a_2$, alors on sait que cette valeur est nécessairement la plus petite de toutes les valeurs de $a_1$ et $a_2$ combinées. On peut donc la placer en première position de la liste à retourner, cette dernière étant alors égale à la concaténation de la première valeur de $a_1$ et du résultat de la fusion entre les valeurs restantes de $a_1$ et toutes celles de $a_2$. Tout ce raisonnement est également valable lorsque les rôles de $a_1$ et $a_2$ sont inversés.

Ce sous-algorithme de fusion, dont on aura noté qu'il est récursif, peut être implémenté comme suit :

```python
def fusionner(a1, a2):
    if not a1:
        return a2
    elif not a2:
        return a1
    elif a1[0] < a2[0]:
        return [a1[0], ] + fusionner(a1[1:], a2)
    else:
        return [a2[0], ] + fusionner(a1, a2[1:])
```

Le plus difficile est fait ; il ne reste plus qu'à écrire la partie du code qui produit deux sous-listes à partir d'une. Cette partie est également récursive.

```python
def tri_fusion(a):
    n = len(a)
    if n <= 1: #cas de base
        return a
    #sinon cas récursif
    a1 = a[0:n//2] #Du debut à la moitié
    a2 = a[n//2:n] #De la moitié à la fin
    return fusionner(tri_fusion(a1), tri_fusion(a2))
```

Il est important de garder en tête que `fusion` travaille sur deux tableaux triés, tandis que `tri_fusion` agit sur des tableaus non triés. Nous donnons ci-dessous un exemple pour le tri d'une petite liste, où la séquence des appels récursifs est affichée et où l'indentation représente la profondeur de la récursion.

```
Tri fusion [5, 1, 3, 2]
    Tri fusion [5, 1]
        Tri fusion [5]
        Tri fusion [1]
        Fusionner [5] [1]
            Fusionner [5] []
    Tri fusion [3, 2]
        Tri fusion [3]
        Tri fusion [2]
        Fusionner [3] [2]
            Fusionner [3] []
    Fusionner [1, 5] [2, 3]
        Fusionner [5] [2, 3]
            Fusionner [5] [3]
                Fusionner [5] []
```

La démonstration de la complexité du tri fusion sort du cadre de ce cours, mais il est possible de montrer qu'elle est $\mathcal{O}(n\log(n))$. Pour s'en convaincre, on peut remarquer qu'il faut $\log(n)$ étapes pour diviser la liste en sous-listes atomiques (nous en avons déjà parlé pour la recherche dichotomique). Or, à chaque étape de cette subdivision, des comparaisons

sont effectuées entre nombres voisins, c'est-à-dire $n$ comparaisons. Nous obtenons donc $n \log(n)$ comparaisons en tout. Durant l'étape de fusion, les nombres suivent en quelques sorte un chemin inverse, ce qui ne change pas la complexité en temps de l'algorithme.

Nous venons de montrer que le tri fusion est plus efficace que le tri par insertion ou le tri à bulles, qui ont une complexité linéaire. Il existe bien d'autres algorithmes de tri. Par exemple, le tri rapide (ou quicksort en anglais) est un algorithme de tri qui a une complexité en $\mathcal{O}(n \log(n))$ en moyenne, mais qui peut avoir une complexité en $\mathcal{O}(n^2)$ dans le pire des cas. Pourtant, le tri rapide est souvent plus efficace que le tri fusion en pratique (à cause des facteurs discutés plus haut, ainsi que pour d'autres raisons propres au fonctionnement des ordinateurs), mais il est plus difficile à implémenter et à analyser.