Polycopié pour le cours de Python (L2 MIASHS)

Romain Tavenard

Chapitre 1

Introduction

Ce document est une tentative de polycopié associé au module de Python pour la deuxième année de licence MIASHS de l'Université de Rennes 2. Il est distribué librement (sous licence CC BY-NC-SA plus précisément) et se veut évolutif, n'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web.

Durant la lecture de ce polycopié, vous trouverez des blocs de code tels que celui-ci :

```
def f(v):
    return v ** 2

x = 5
y = f(3 * x + 2)
print(y)
# [Sortie] 77
```

Nous prendrons notamment l'habitude de reporter les valeurs affichées par l'exécution du programme considéré dans un terminal avec la syntaxe utilisée à la dernière ligne du code ci-dessus.

Dans ce document, nous allons donc nous intéresser au langage Python. Pour tester les exemples présentés au fil de ce document ou réaliser les exercices proposés, vous aurez deux possibilités. La première consiste à ouvrir une **console Python**, à l'aide de la commande suivante (si vous êtes sous Unix, en supposant que le symbole `$` corresponde au prompt de votre *shell*) :

```
$ python
Python 3.5.1 (default, Dec  9 2015, 11:28:16)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.1.76)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Lors de l'exécution de cette commande, on peut remarquer plusieurs choses. Tout d'abord, au démarrage, la console Python nous indique la version de Python qui est exécutée. Cela est important, car il existe notamment une importante différence entre les versions 2 (2.x.y) et 3 (3.x.y) de Python. Dans ce document, nous supposons l'utilisation de Python dans sa version 3, comme dans la console affichée plus haut. Enfin, une fois la console démarée, on voit apparaître un prompt Python (`>>>`) qui indique que vous pouvez, à partir de ce point, entrer du code Python et en demander l'exécution en appuyant sur la touche *retour chariot* (ou "Entrée") de votre clavier.

L'autre façon de programmer en Python, plus adaptée dès lors que l'on souhaite conserver une trace de ce qu'on a écrit, consiste à enregistrer vos commandes dans un fichier texte (en respectant la convention qui consiste à utiliser l'extension `.py` pour le nom de fichier) puis à faire exécuter votre programme par l'interpréteur Python :

```
$ python nom_de_mon_fichier.py
[...]
```

Chapitre 2

Structures de données et structures de contrôle

Dans ce chapitre, nous allons nous intéresser aux éléments de base de la syntaxe Python : les structures de données d'une part et les structures de contrôle d'autre part. Les structures de données vont nous permettre de stocker dans la mémoire de l'ordinateur (dans le but de les traiter ensuite) des données tandis que les structures de contrôle vont servir à définir nos interactions avec ces données.

2.1 Variables

En Python, les données sont stockées dans des variables, on ne peut pas définir de constante. Une variable est une association entre un symbole (le nom de la variable) et une valeur, cette dernière pouvant varier au cours de l'exécution du programme. Les variables Python sont typées dynamiquement, ce qui signifie qu'une variable, à un moment donné de l'exécution d'un programme, a un type précis qui lui est attribué, mais que celui-ci peut évoluer au cours de l'exécution du programme.

2.1.1 Types des variables Python

Les types de base existant en Python sont les suivants :

- `int` : entier ;
- `float` : nombre à virgule ;
- `complex` : nombre complexe (peu utilisé en pratique) ;
- `str` : chaîne de caractères ;
- `bool` : booléen (pouvant prendre les valeurs `True` ou `False`).

De plus, il existe un type spécial (`NoneType`) ne permettant qu'une seule valeur : la valeur `None` qui signifie "pas de valeur" ou "valeur manquante".

En Python, le type d'une variable n'est pas déclaré par l'utilisateur : il est défini par l'usage (la valeur effective que l'on décide de stocker dans la variable en question).

Par exemple, l'instruction suivante en Python attribue la valeur `12` à la variable `v`, qui devient donc automatiquement de type entier :

```
v = 12
```

Ainsi, les instructions suivantes ont toutes une incidence sur le type des variables considérées :

```
v = 12     # v est alors de type entier
c = "abc"  # c est de type chaîne de caractères
d = 'abc'  # d est également de type chaîne de caractères
           # les contenus de c et d sont identiques
v = 12.    # v change de type et est désormais de type nombre à virgule
```

Pour vérifier le type d'une variable, il suffit d'utiliser la fonction `type` de la librairie standard :

```
print(type(v))  # la fonction print(.) permet d'afficher
                # une information dans le terminal
# [Sortie] <class 'float'>
```

2.1.2 Opération d'assignation

Comme le montrent les exemples précédents, pour pouvoir utiliser des variables, on doit leur donner un nom (placé à gauche du signe égal dans l'opération d'affectation). Ces noms de variables doivent respecter certaines contraintes :

- ils doivent débuter par une lettre (minuscule ou majuscule, peu importe) ou par le symbole `_` ;
- ils ne doivent contenir que des lettres, des chiffres et des symboles `_` ;
- ils ne doivent pas correspondre à un quelconque mot réservé du langage Python, dont voici la liste :

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

- ils ne doivent pas correspondre à des noms de fonction de la librairie standard de Python (cette dernière condition n'est en fait qu'une bonne pratique à observer) : vous apprendrez au fur et à mesure les noms de ces fonctions.

Les noms de variable en Python sont sensibles à la casse, ainsi les variables `maVariable` et `mavariable` ne pointent pas sur les mêmes données en mémoire. Pour s'en convaincre, on peut exécuter le code suivant :

```
mavariable = 12
maVariable = 15
print(mavariable)
# [Sortie] 12
print(maVariable)
# [Sortie] 15
```

Comme on l'a vu plus haut, on utilise en Python l'opérateur = pour assigner une valeur à une variable. La sémantique de cet opérateur est la suivante : "assigner la valeur contenue dans le membre de droite à la variable du membre de gauche". Ainsi, il est tout à fait valide d'écrire, en Python :

```
x = 3.9 * x * (1 - x)
```

Pour exécuter cette instruction, l'interpréteur Python commencera par évaluer le membre de droite en utilisant la valeur courante de la variable `x`, puis affectera la valeur correspondant au résultat de l'opération `3.9 * x * (1 - x)` dans la variable `x`.

2.1.3 Opérateurs et priorité

On le voit dans l'exemple précédent, pour manipuler des variables, on utilisera des opérateurs (dont les plus connus sont les opérateurs arithmétiques). Le tableau suivant dresse une liste des opérateurs définis pour les variables dont le type est l'un des types numériques (entier, nombre à virgule, nombre complexe) :

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
**	Élévation à la puissance
%	Modulo (non défini pour les nombres complexes)

De plus, pour chacun de ces opérateurs, il existe un opérateur associé qui réalise successivement l'opération demandée puis l'affectation de la nouvelle valeur à la variable en question. Ainsi, l'instruction suivante :

```
x = x + 2
```

qui ajoute 2 à la valeur courante de `x` puis stocke le résultat du calcul dans `x` peut se réécrire :

```
x += 2
```

Ceci est purement un raccourci de notation, s'il ne vous semble pas évident à maîtriser au premier abord, vous pouvez vous en passer et toujours utiliser la notation `x = x + 2`.

Enfin, lorsque l'évaluation d'une expression implique plusieurs opérateurs, les règles de priorité sont les suivantes (de la priorité maximale à la priorité minimale) :

1. parenthèses ;
2. élévation à la puissance ;
3. multiplication / division ;
4. addition / soustraction ;
5. de gauche à droite.

Pour prendre un exemple concret, pour évaluer l'expression :

```
3.9 * x * (1 - x)
```

on commencera par évaluer le contenu de la parenthèse puis, les 2 opérations restantes étant toutes des multiplications, on les effectuera de gauche à droite.

De plus, lorsqu'une opération est effectuée entre deux variables de types différents, le type le plus générique est retenu. Par exemple, si l'on multiplie un entier par un nombre à virgule, le résultat sera de type `float`. De même, le résultat de l'addition entre un nombre complexe et un nombre à virgule est un complexe.

Attention. Comme indiqué en introduction, ce polycopié suppose que vous utilisez Python dans sa version 3. Il est à noter qu'il existe une différence importante entre Python 2 et Python 3 dans la façon d'effectuer des opérations mêlant nombres entiers et flottants. Par exemple, l'opération suivante :

```
x = 2 / 3
```

stockera, en Python 2, la valeur 0 (résultat de la division **entière** de 2 par 3) dans la variable `x` alors qu'en Python 3, la division **flottante** sera effectuée et ainsi `x` contiendra `0.666666...` En Python 3, si l'on souhaite effectuer une division entière, on pourra utiliser l'opérateur `//` :

```
print(2 // 3)
# [Sortie] 0
```

2.2 Structures de contrôle

Un programme est une séquence d'instructions dont l'ordre doit être respecté. Au-delà de cet aspect séquentiel, on peut souhaiter :

— n'effectuer certaines instructions que si une condition est vérifiée ;

— répéter certaines instructions ;
— factoriser une sous-séquence d'instructions au sein d'une fonction pour pouvoir y faire appel à plusieurs reprises dans le programme.

Les structures de contrôle associées à ces différents comportements sont décrits dans la suite de cette section.

2.2.1 Structures conditionnelles

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

```
x = 12
if x > 0:
    print("X est positif")
    x = 4
# [Sortie] X est positif
```

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

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

```
x = -1
if x > 0:
    print("X est positif")
    x = 4
else:
    print("X est négatif")
# [Sortie] X est négatif
```

Là encore, on remarque que l'indentation est de rigueur pour chacun des deux blocs d'instructions. On note également que le mot-clé **else** se trouve au même niveau que le **if** auquel il se réfère.

Enfin, de manière plus générale, il est possible de définir plusieurs comportements en fonction de plusieurs tests successifs, à l'aide du mot-clé **elif** :

```
x = -1
if x > 0:
    print("X est positif")
```

```
    x = 4
elif x > -2:
    print("X est compris entre -2 et 0")
elif x > -4:
    print("X est compris entre -4 et -2")
else:
    print("X est inférieur à -4")
# [Sortie] X est compris entre -2 et 0
```

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

Opérateur	Comparaison effectuée	Exemple
`<`	Plus petit que	`x < 0`
`>`	Plus grand que	`x > 0`
`<=`	Plus petit ou égal à	`x <= 0`
`>=`	Plus grand ou égal à	`x >= 0`
`==`	Égal à	`x == 0`
`!=`	Différent de	`x != 0`
`is`	Test d'égalité pour le cas de la valeur `None`	`x is None`
`is not`	Test d'inégalité pour le cas de la valeur `None`	`x is not None`
`in`	Test de présence d'une valeur dans une liste	`x in [1, 5, 7]`

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

2.2.2 Boucles

Il existe, en Python comme dans une grande majorité des langages de programmation, deux types de boucles :

- les boucles qui s'exécutent tant qu'une condition est vraie ;
- les boucles qui répètent la même série d'instructions pour différentes valeurs d'une variable (appelée **variable de boucle**).

2.2.2.1 Boucles `while`

Les premières ont une syntaxe très similaire à celle des structures conditionnelles simples :

```
x = 0
while x <= 10:
```

```
    print(x)
    x = 2 * x + 2
# [Sortie] 0
# [Sortie] 2
# [Sortie] 6
```

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

```
x = 2
y = 0
while x > 0:
    y = y - 1
y = 2
```

En effet, on a ici une boucle qui s'exécutera tant que `x` est positif, or la valeur de cette variable est initialisée à 2 et n'est pas modifiée au sein de la boucle, la condition sera donc toujours vérifiée et le programme ne sortira jamais de la boucle.

2.2.2.2 Boucles `for`

Le second type de boucle repose en Python sur l'utilisation de listes (ou, plus généralement, d'itérables) dont nous reparlerons plus en détail dans la suite de cet ouvrage. Sachez pour le moment qu'une liste est un ensemble ordonné d'éléments. On peut alors exécuter une série d'instructions pour toutes les valeurs d'une liste :

```
for x in [1, 5, 7]:
    print(x)
y = 2
# [Sortie] 1
# [Sortie] 5
# [Sortie] 7
```

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

2.2.3 Fonctions

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

```
print(x)
```

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

```
x = type(1)  # On stocke dans x la valeur retournée par type
y = type(2.)
if x == y:
    z = 1
else:
    z = 2
```

2.2.3.1 Définition d'une fonction

Lorsqu'un ensemble d'instructions est susceptible d'être utilisé à plusieurs occasions dans un ou plusieurs programmes, il est recommandé de l'isoler au sein d'une fonction. Cela présentera les avantages suivants :

- en donnant un nom à la fonction et en listant la liste de ses arguments, on explicite la sémantique de l'ensemble d'instructions en question, ses entrées et sorties éventuelles, ce qui rend le code beaucoup plus lisible ;
- s'il est nécessaire d'adapter à l'avenir le code pour résoudre un *bug* ou le rendre plus générique, vous n'aurez à modifier le code qu'à un endroit (dans le corps de la fonction) et non pas à chaque fois que le code est répété.

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

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

On a ici défini une fonction

- dont le nom est `f` ;
- qui prend un seul argument, noté `x` ;
- qui retourne une valeur, comme indiqué dans la ligne débutant par le mot-clé `return`.

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

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

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

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

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

Les trois appels suivants sont équivalents :

```
affiche_infos_personne(80, 180)
# [Sortie] Poids: 80
# [Sortie] Taille: 180
affiche_infos_personne(taille=180, poids=80)
# [Sortie] Poids: 80
# [Sortie] Taille: 180
affiche_infos_personne(poids=80, taille=180)
# [Sortie] Poids: 80
# [Sortie] Taille: 180
```

Évidemment, pour que cela soit vraiment utile, il est hautement recommandé d'utiliser des noms d'arguments explicites lors de la définition de vos fonctions.

2.2.3.2 Argument(s) optionnel(s) d'une fonction

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

Par exemple, la fonction `print` dispose de plusieurs arguments facultatifs, comme le caractère par lequel terminer l'affichage (par défaut, un retour à la ligne, `"\n"`) :

```
print("La vie est belle")
# [Sortie] La vie est belle
print("Life is beautiful")
```

```
# [Sortie] Life is beautiful
print("La vie est belle", end="--")
print("Life is beautiful", end="--")
# [Sortie] La vie est belle--Life is beautiful--
```

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

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

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

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

2.3 Les modules en Python

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

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

```
import math

print(math.sin(0))
# [Sortie] 0.0
print(math.pi)
# [Sortie] 3.141592653589793
print(math.cos(2 * math.pi))
# [Sortie] 1.0
print(math.sqrt(2))
# [Sortie] 1.4142135623730951
```

Vous remarquerez ici que l'instruction d'import du module se trouve nécessairement avant les instructions faisant référence aux fonctions et variables de ce module, faute de quoi ces dernières ne seraient pas définies. De manière générale,

vous prendrez la bonne habitude d'écrire les instructions d'import en tout début de vos fichiers Python, pour éviter tout souci.

Exercice 2.1 Écrivez une fonction en Python qui prenne en argument une longueur `l` et retourne l'aire du triangle équilatéral de côté `l`.

Chapitre 3

Les listes

En Python, les listes stockent des séquences d'éléments. Il n'est pas nécessaire que tous les éléments d'une liste soient du même type, même si dans les exemples que nous considérerons, ce sera souvent le cas.

On peut trouver des informations précieuses sur le sujet des listes dans l'aide en ligne de Python disponible à l'adresse : https://docs.python.org/3/tutorial/datastructures.html.

3.1 Avant-propos : listes et itérables

Dans la suite, nous parlerons de listes, qui est un type de données bien spécifique en Python. Toutefois, une grande partie de notre propos pourra se transposer à l'ensemble des itérables en Python (c'est-à-dire l'ensemble des objets Python dont on peut parcourir les éléments un à un).

Il existe toutefois une différence majeure entre listes et itérables : nous verrons dans la suite de ce chapitre que l'on peut accéder au i-ème élément d'une liste simplement, alors que ce n'est généralement pas possible pour un itérable (pour ce dernier, il faudra parcourir l'ensemble de ses éléments et s'arrêter lorsque l'on est effectivement rendu au i-ème).

Toutefois, si l'on a un itérable `iterable`, il est possible de le transformer en liste simplement à l'aide de la fonction `list` :

```
l = list(iterable)
```

3.2 Création de liste

Pour créer une liste contenant des éléments définis (par exemple la liste contenant les entiers 1, 5 et 7), il est possible d'utiliser la syntaxe suivante :

```
liste = [1, 5, 7]
```

De la même façon, on peut créer une liste vide (ne contenant aucun élément) :

```
liste = []
print(len(liste))
# [Sortie] 0
```

On voit ici la fonction `len` qui retourne la taille d'une liste passée en argument (ici 0 puisque la liste est vide).

Toutefois, lorsque l'on souhaite créer des listes longues (par exemple la liste des 1000 premiers entiers), cette méthode est peu pratique. Heureusement, il existe des fonctions qui permettent de créer de telles listes. Par exemple, la fonction `range(a, b)` retourne un itérable contenant les entiers de `a` (inclus) à `b` (exclu) :

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

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

3.3 Accès aux éléments d'une liste

Pour accéder au i-ème élément d'une liste, on utilise la syntaxe :

```
l[i]
```

Attention, toutefois, le premier indice d'une liste est 0, on a donc :

```
l = [1, 5, 7]
print(l[1])
print(l[0])
# [Sortie] 5
# [Sortie] 1
```

On peut également accéder au dernier élément d'une liste en demandant l'élément d'indice `-1` :

```
l = [1, 5, 7]
print(l[-1])
```

```
print(l[-2])
print(l[-3])
# [Sortie] 7
# [Sortie] 5
# [Sortie] 1
```

De la même façon, on peut accéder au deuxième élément en partant de la fin *via* l'indice `-2`, *etc.*

Ainsi, pour une liste de taille n, les valeurs d'indice valides sont les entiers compris entre $-n$ et $n-1$ (inclus).

Il est également à noter que l'accès aux éléments d'une liste peut se faire en lecture (lire l'élément stocké à l'indice `i`) comme en écriture (modifier l'élément stocké à l'indice `i`) :

```
l = [1, 5, 7]
print(l[1])
l[1] = 2
print(l)
# [Sortie] 5
# [Sortie] [1, 2, 7]
```

Enfin, on peut accéder à une sous-partie d'une liste à l'aide de la syntaxe `l[d:f]` où `d` est l'indice de début et `f` est l'indice de fin (exclu). Ainsi, on a :

```
l = [1, 5, 7, 8, 0, 9, 8]
print(l[2:4])
# [Sortie] [7, 8]
```

Lorsque l'on utilise cette syntaxe, si l'on omet l'indice de début, la sélection commence au début de la liste et si l'on omet l'indice de fin, elle s'étend jusqu'à la fin de la liste :

```
l = [1, 5, 7, 8, 0, 9, 8]
print(l[:3])
# [Sortie] [1, 5, 7]
print(l[5:])
# [Sortie] [9, 8]
```

3.4 Parcours d'une liste

Lorsque l'on parcourt une liste, on peut vouloir accéder :

— aux éléments stockés dans la liste uniquement ;
— aux indices de la liste uniquement (même si c'est rare) ;
— aux indices de la listes et aux éléments associés.

Ces trois cas de figure impliquent trois parcours de liste différents, décrits dans ce qui suit.

Attention. Quel que soit le parcours de liste utilisé, il est fortement déconseillé de supprimer ou d'insérer des éléments dans une liste pendant le parcours de celle-ci.

3.4.1 Parcours des éléments

Pour parcourir les éléments d'une liste, on utilise une boucle `for` :

```
l = [1, 5, 7]
for elem in l:
    print(elem)
# [Sortie] 1
# [Sortie] 5
# [Sortie] 7
```

3.4.2 Parcours par indices

Pour avoir accès aux indices (positifs) de la liste, on devra utiliser un subterfuge. On sait que les indices d'une liste sont les entiers compris entre 0 (inclus) et la taille de la liste (exclu). On va donc utiliser la fonction `range` pour cela :

```
l = [1, 5, 7]
n = len(l)  # n = 3 ici
for i in range(n):
    print(i)
# [Sortie] 0
# [Sortie] 1
# [Sortie] 2
```

3.4.3 Parcours par éléments et indices

Dans certains cas, enfin, on a besoin de manipuler simultanément les indices d'une listes et les éléments associés. Cela se fait à l'aide de la fonction `enumerate` :

```
l = [1, 5, 7]
for i, elem in enumerate(l):
    print(i, elem)
# [Sortie] 0 1
# [Sortie] 1 5
# [Sortie] 2 7
```

On a donc ici une boucle `for` pour laquelle, à chaque itération, on met à jour les variables `i` (qui contient l'indice courant) et `elem` (qui contient l'élément se trouvant à l'indice `i` dans la liste `l`).

Exercice 3.1 Écrivez une fonction en Python qui permette de calculer l'argmax d'une liste, c'est-à-dire l'indice auquel est stockée la valeur maximale de la liste.

3.5 Manipulations de listes

Nous présentons dans ce qui suit les opérations élémentaires de manipulation de listes.

3.5.1 Insertion d'élément

Pour insérer un nouvel élément dans une liste, on peut :

- rajouter un élément à la fin de la liste à l'aide de la méthode `append` ;
- insérer un élément à l'indice `i` de la liste à l'aide de la méthode `insert`.

Comme vous pouvez le remarquer, il est ici question de méthodes et non plus de fonctions. Pour l'instant, sachez que les méthodes sont des fonctions spécifiques à certains objets, comme les listes par exemples. L'appel de ces méthodes est un peu particulier, comme vous pouvez le remarquer dans ce qui suit :

```
l = [1, 5, 7]
l.append(2)
print(l)
# [Sortie] [1, 5, 7, 2]
l.insert(2, 0)  # insère la valeur 0 à l'indice 2
print(l)
# [Sortie] [1, 5, 0, 7, 2]
```

3.5.2 Suppression d'élément

Si l'on souhaite, maintenant, supprimer un élément dans une liste, deux cas de figures peuvent se présenter. On peut souhaiter :

- supprimer l'élément situé à l'indice `i` dans la liste, à l'aide de la méthode `pop` ;
- supprimer la première occurrence d'une valeur donnée dans la liste à l'aide de la méthode `remove`.

```
l = [1, 5, 7]
l.pop(1)  # l'élément d'indice 1 est le deuxième élément de la liste !
print(l)
```

```
# [Sortie] [1, 7]
l.pop()  # par défaut, supprime le dernier élément de la liste
print(l)
# [Sortie] [1]
l = [7, 5, 1]
l.remove(1)
print(l)
# [Sortie] [7, 5]
```

On peut noter que la méthode `pop` retourne la valeur supprimée, ce qui peut s'avérer utile :

```
l = [1, 5, 7]
v = l.pop(1)
print(v)
print(l)
# [Sortie] 5
# [Sortie] [1, 7]
```

3.5.3 Recherche d'élément

Pour trouver l'indice de la première occurrence d'une valeur dans une liste, on utilisera la méthode `index` :

```
l = [1, 5, 7]
print(l.index(7))
# [Sortie] 2
```

Si l'on ne cherche pas à connaître la position d'une valeur dans une liste mais simplement à savoir si une valeur est présente dans la liste, on peut utiliser le mot-clé `in` :

```
l = [1, 5, 7]
if 5 in l:
    print("5 est dans l")
# [Sortie] 5 est dans l
```

3.5.4 Création de listes composites

On peut également concaténer deux listes (c'est-à-dire mettre bout à bout leur contenu) à l'aide de l'opérateur `+` :

```
l1 = [1, 5, 7]
l2 = [3, 4]
l = l1 + l2
print(l)
# [Sortie] [1, 5, 7, 3, 4]
```

Dans le même esprit, l'opérateur $*$ peut aussi être utilisé pour des listes :

```
l1 = [1, 5]
l2 = 3 * l1
print(l2)
# [Sortie] [1, 5, 1, 5, 1, 5]
```

Bien entendu, vu le sens de cet opérateur, on ne peut multiplier une liste que par un entier.

3.5.5 Tri de liste

Enfin, on peut trier les éléments contenus dans une liste à l'aide de la fonction `sorted` :

```
l = [4, 5, 2]
l2 = sorted(l)
print(l2)
# [Sortie] [2, 4, 5]
```

3.6 Copie de liste

Pour la plupart des variables, en Python, la copie ne pose pas de problème :

```
a = 12
b = a
a = 5
print(a, b)
# [Sortie] 5 12
```

Cela ne se passe pas de la même façon pour les listes. En effet, si `l` est une liste, lorsque l'on écrit :

```
l2 = l
```

on ne recopie pas le contenu de `l` dans `l2`, mais on crée une variable `l2` qui va "pointer" vers la même position dans la mémoire de votre ordinateur que `l`. La différence peut sembler mince, mais cela signifie que si l'on modifie `l` même après l'instruction `l2 = l`, la modification sera répercutée sur `l2` :

```
l = [1, 5, 7]
l2 = l
l[1] = 2
print(l, l2)
# [Sortie] [1, 2, 7] [1, 2, 7]
```

Lorsque l'on souhaite éviter ce comportement, il faut effectuer une copie explicite de liste, à l'aide par exemple de la fonction `list` :

```
l = [1, 5, 7]
l2 = list(l)
l[1] = 2
print(l, l2)
# [Sortie] [1, 2, 7] [1, 5, 7]
```

3.7 Bonus : listes en compréhension

Il est possible de créer des listes en filtrant et/ou modifiant certains éléments d'autres listes ou itérables. Supposons par exemple que l'on souhaite créer la liste des carrés des 10 premiers entiers naturels. Le code qui suit présente deux façons équivalentes de créer une telle liste :

```
# Façon "classique"
l = []
for i in range(10):
    l.append(i ** 2)
print(l)
# [Sortie] [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# En utilisant les listes en compréhension
l = [i ** 2 for i in range(10)]
print(l)
# [Sortie] [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On remarque que la syntaxe de liste en compréhension est plus compacte. On peut également appliquer un filtre sur les éléments de la liste de départ (ici `range(10)`) à considérer à l'aide du mot-clé `if` :

```
l = [i ** 2 for i in range(10) if i % 2 == 0]
print(l)
# [Sortie] [0, 4, 16, 36, 64]
```

Ici, on n'a considéré que les entiers pairs.

Chapitre 4

Les chaînes de caractères

Nous nous intéressons maintenant à un autre type de données particulier du langage Python : les chaînes de caractères (type `str`). Pour créer une chaîne de caractères, il suffit d'utiliser des guillemets, simples ou doubles (les deux sont équivalents) :

```
s1 = "abc"
s2 = 'bde'
```

Comme pour les listes (et peut-être même plus encore), il est fortement conseillé de se reporter à l'aide en ligne dédiée lorsque vous avez des doutes sur la manipulation de chaînes de caractères : https://docs.python.org/3/library/stdtypes.html#string-methods

4.1 Conversion d'une chaîne en nombre

Si une chaîne de caractères représente une valeur numérique (comme la chaîne `"10.2"` par exemple), on peut la transformer en un entier ou un nombre à virgule, afin de l'utiliser ensuite pour des opérations arithmétiques :

```
s = '10.2'
f = float(s)
print(f)
# [Sortie] 10.2
print(f == s)
# [Sortie] False
print(f + 2)
# [Sortie] 12.2
```

```
s = '10'
i = int(s)
```

```
print(i)
# [Sortie] 10
print(i == s)
# [Sortie] False
print(i - 1)
# [Sortie] 9
```

4.2 Analogie avec les listes

Les chaînes de caractères se manipulent en partie comme des listes. On peut ainsi obtenir la taille d'une chaîne de caractères à l'aide de la fonction `len`, ou accéder à la i-ème lettre d'une chaîne de caractères avec la notation `s[i]`. Comme pour les listes, il est possible d'indicer une chaîne de caractères en partant de la fin, en utilisant des indices négatifs :

```
s = "abcdef"
print(s[0])
# [Sortie] a
print(s[-1])
# [Sortie] f
```

De même, on peut sélectionner des sous-parties de chaînes de caractères à partir des indices de début et de fin de la sélection. Comme pour les listes, l'indice de fin correspond au premier élément exclu de la sélection :

```
s = "abcdef"
print(s[2:4])
# [Sortie] cd
```

Comme pour les listes, on peut concaténer deux chaînes de caractères à l'aide de l'opérateur `+` ou répéter une chaîne de caractères avec l'opérateur `*` :

```
s = "ab" + ('cde' * 3)
print(s)
# [Sortie] abcdecdecde
```

On peut également tester la présence d'une sous-chaîne de caractères dans une chaîne avec le mot-clé `in` :

```
s = "abcde"
print("a" in s)
# [Sortie] True
print("bcd" in s)
# [Sortie] True
print("bCd" in s)
# [Sortie] False
```

Attention. Toutefois, l'analogie entre listes et chaînes de caractères est loin d'être parfaite. Par exemple, on peut accéder au i-ème élément d'une liste en lecture, mais pas en écriture. Si `s` est une chaîne de caractères, on ne peut pas exécuter `s[2] = "c"` par exemple.

4.3 Principales méthodes de la classe `str`

La liste de méthodes de la classe `str` qui suit n'est pas exhaustive, il est conseillé de consulter l'aide en ligne de Python pour plus d'informations.

- `ch.count(sub)`: Retourne le nombre d'occurrences de `sub` dans `ch`
- `ch.endswith(suffix)`: Retourne `True` si `ch` se termine par `suffix`
- `ch.startswith(prefix)`: Retourne `True` si `ch` commence par `suffix`
- `ch.find(sub)`: Retourne l'indice du début de la première occurrence de `sub` dans `ch`
- `ch.rfind(sub)`: Retourne l'indice du début de la dernière occurrence de `sub` dans `ch`
- `ch.islower()`: Retourne `True` si `ch` est constituée uniquement de caractères minuscules
- `ch.isupper()`: Retourne `True` si `ch` est constituée uniquement de caractères majuscules
- `ch.isnumeric()`: Retourne `True` si `ch` est constituée uniquement de chiffres
- `ch.lower()`: Retourne la version minuscule de `ch`
- `ch.upper()`: Retourne la version majuscule de `ch`
- `ch.replace(old, new)`: Retourne une copie de `ch` dans laquelle la *première* occurrence de `old` a été remplacée par `new`
- `ch.split(sep=None)`: Retourne une liste contenant des morceaux de `ch` découpée à chaque occurrence de `sep` (n'importe quel espace par défaut)
- `ch.strip()`: Retourne une version "nettoyée" de `ch` dans laquelle on a enlevé tous les espaces en début et en fin de chaîne

Exercice 4.1 Écrivez une fonction qui prenne en argument deux chaînes de caractères `s` et `prefix` et retourne le nombre de mots de la chaîne `s` qui débutent par la chaîne `prefix`.

Chapitre 5

Les dictionnaires

Comme une liste, un dictionnaire est une collection de données. Mais, à la différence des listes, les dictionnaires ne sont pas ordonnés (ou, tout du moins, ils le sont dans un ordre qui ne nous est pas naturel). Chaque entrée dans un dictionnaire est une association entre une **clé** (équivalente à un indice pour une liste) et une **valeur**. Alors que les indices d'une liste sont forcément les entiers compris entre 0 et la taille de la liste exclue, les clés d'un dictionnaire sont des valeurs quelconques, la seule contrainte étant qu'on ne peut pas avoir deux fois la même clé dans un dictionnaire. Notamment, ces clés ne sont pas nécessairement des entiers, on utilisera en effet souvent des dictionnaires lorsque l'on souhaite stocker des valeurs associées à des chaînes de caractères (qui seront les clés du dictionnaire).

Pour définir un dictionnaire par ses paires clés-valeurs en Python, on peut utiliser la syntaxe suivante :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico)
# [Sortie] {'a': 123, 'bbb': None, 'z': 7}
```

On remarque ici que l'ordre dans lequel on a entré des paires clés/valeurs n'est pas conservé lors de l'affichage.

5.1 Modification du contenu d'un dictionnaire

Pour modifier la valeur associée à une clé d'un dictionnaire, la syntaxe est similaire à celle utilisée pour les listes, en remplaçant les indices par les clés :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
mon_dico["a"] = 1000
```

```
print(mon_dico)
# [Sortie] {'a': 1000, 'bbb': None, 'z': 7}
```

De même, on peut créer une nouvelle paire clé/valeur en utilisant la même syntaxe :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
mon_dico["c"] = -1
print(mon_dico)
# [Sortie] {'c': -1, 'a': 123, 'bbb': None, 'z': 7}
```

Enfin, pour supprimer une paire clé/valeur d'un dictionnaire, on utilise le mot-clé `del` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
del mon_dico["a"]
print(mon_dico)
# [Sortie] {'bbb': None, 'z': 7}
```

5.2 Lecture du contenu d'un dictionnaire

Pour lire la valeur associée à une clé du dictionnaire, on peut utiliser la même syntaxe que pour les listes :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico["a"])
# [Sortie] 123
```

Par contre, si la clé demandée n'existe pas, cela génèrera une erreur. Pour éviter cela, on peut utiliser la méthode `get` qui permet de définir une valeur par défaut à retourner si la clé n'existe pas :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
print(mon_dico.get("a", 0))
# [Sortie] 123
print(mon_dico.get("b", 0))
# [Sortie] 0
```

5.3 Parcours d'un dictionnaire

Pour parcourir le contenu d'un dictionnaire, il existe, comme pour les listes, trois possibilités.

5.3.1 Parcours par valeurs

Si l'on souhaite uniquement accéder aux valeurs stockées dans le dictionnaire, on utilisera la méthode `values` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for val in mon_dico.values():
    print(val)
# [Sortie] 123
# [Sortie] None
# [Sortie] 7
```

5.3.2 Parcours par clés

Si l'on souhaite uniquement accéder aux clés stockées dans le dictionnaire, on utilisera la méthode `keys` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for cle in mon_dico.keys():
    print(cle)
# [Sortie] a
# [Sortie] bbb
# [Sortie] z
```

5.3.3 Parcours par couples clés/valeurs

Si l'on souhaite accéder simultanément aux clés stockées dans le dictionnaire et aux valeurs associées, on utilisera la méthode `items` :

```
mon_dico = {"a" : 123, "z" : 7, "bbb" : None}
for cle, valeur in mon_dico.items():
    print(cle, valeur)
# [Sortie] a 123
# [Sortie] bbb None
# [Sortie] z 7
```

Exercice 5.1 Écrivez un programme Python qui compte le nombre d'occurrences de chacun des mots d'une chaîne de caractères et stocke le résultat dans un dictionnaire :

```
s = "la vie est belle c'est la vie"
# [...]
print(d)
# [Sortie] {"c'est": 1, 'la': 2, 'belle': 1, 'est': 1, 'vie': 2}
```

Chapitre 6

Lecture et écriture de fichiers textuels

Dans ce chapitre, nous nous intéressons à la lecture/écriture de fichiers textuels par un programme Python. Un premier élément qu'il est nécessaire de maîtriser pour lire ou écrire des fichiers textuels est la notion d'encodage. Il faut savoir qu'il existe plusieurs façons d'encoder un texte. Nous nous focaliserons ici sur les deux encodages que vous êtes les plus susceptibles de rencontrer (mais sachez qu'il en existe bien d'autres) :

— l'encodage Unicode 8 bits (UTF-8), dont le code en python est `"utf-8"` ;
— l'encodage Latin-1 (ISO-8859-1) dont le code en python est `"iso-8859-1"`.

La principale différence entre ces deux encodage réside dans leur façon de coder les accents. Ainsi, si le texte que vous lisez/écrivez ne contient aucun accent ou caractère spécial, il est probable que la question de l'encodage ne soit pas problématique dans votre cas. Au contraire, s'il est possible que vous utilisiez de tels caractères, il faudra bien faire attention à l'encodage utilisé, que vous spécifierez à l'ouverture du fichier. Si votre programme doit lire un fichier, il faudra donc vous assurer de l'encodage associé à ce fichier (en l'ouvrant par exemple avec un éditeur de texte qui soit suffisamment avancé pour vous fournir cette information). Si vous écrivez un programme qui écrit un fichier, il faudra vous poser la question de l'utilisation future qui sera faite de ce fichier : s'il est amené à être ouvert par un autre utilisateur, il serait pertinent de vous demander quel encodage sera le moins problématique pour cet utilisateur, par exemple.

Si vous n'avez pas de contrainte extérieure pour ce qui est de l'encodage, vous utiliserez l'encodage UTF-8 par défaut.

6.1 Lecture de fichiers textuels

Ce que nous apellons lecture de fichiers textuels en Python consiste à copier le contenu d'un fichier dans une (ou plusieurs) chaîne(s) de caractères. Cela implique deux étapes en Python :

1. ouvrir le fichier en lecture ;
2. parcourir le contenu du fichier.

La première étape d'ouverture du fichier en lecture est commune à tous les types de fichiers textuels. En supposant que le nom du fichier à ouvrir soit stocké sous forme de chaîne de caractères dans la variable `nom_fichier`, le code suivant ouvre un fichier en lecture avec l'encodage UTF-8 et stocke dans la variable `fp` un pointeur sur l'endroit où nous sommes rendus dans notre lecture du fichier (pour l'instant, le début du fichier) :

```
fp = open(nom_fichier, "r", encoding="utf-8")
```

Le second argument (`"r"`) indique que le fichier doit être ouvert en mode *read*, donc en lecture.

6.1.1 Fichiers textuels génériques

Une fois le fichier ouvert en lecture, on peut le lire ligne par ligne à l'aide de la boucle suivante :

```
fp = open(nom_fichier, "r", encoding="utf-8")
for ligne in fp.readlines():
    print(ligne)
```

Ici, la variable `ligne`, de type chaîne de caractères, contiendra successivement le texte de chacune des lignes du fichier considéré.

6.1.2 Fichiers *Comma-Separated Values* (CSV)

Les fichiers *Comma-Separated Values* (CSV) permettent de stocker des données organisées sous la forme de tableaux dans des fichiers textuels. À l'origine, ces fichiers étaient organisées par ligne et au sein de chaque ligne les cellules du tableau (correspondant aux différentes colonnes) étaient séparées par des virgules (d'où le nom de ce type de fichiers). Aujourd'hui, la définition de ce format (lien) est plus générale que cela et différents délimiteurs sont acceptés. Pour manipuler ces fichiers, il existe en Python un module dédié, appelé `csv`. Ce module contient notamment une fonction `reader` permettant de simplifier la lecture de fichiers CSV. La syntaxe d'utilisation de cette fonction est la suivante (vous remarquerez la présence de l'attribut `delimiter`) :

```
import csv

nom_fichier = "..." # À remplacer par le chemin vers le fichier :)

# Contenu supposé du fichier :
# 1;2;3
# a;b

fp = open(nom_fichier, "r", encoding="utf-8")
for ligne in csv.reader(fp, delimiter=";"):
    for cellule in ligne:
        print(cellule)
    print("Fin de ligne")
# [Sortie] 1
# [Sortie] 2
# [Sortie] 3
# [Sortie] Fin de ligne
# [Sortie] a
# [Sortie] b
# [Sortie] Fin de ligne
```

On remarque ici que, contrairement au cas de fichiers textuels génériques, la variable de boucle `ligne` n'est plus une chaîne de caractères mais une liste de chaînes de caractères. Les éléments de cette liste sont les cellules du tableau représenté par le fichier CSV.

6.2 Écriture de fichiers textuels

Ce que nous apellons écriture de fichiers textuels en Python consiste à copier le contenu d'une (ou plusieurs) chaîne(s) de caractères dans un fichier. Cela implique deux étapes en Python :

1. ouvrir le fichier en écriture ;
2. ajouter du contenu dans le fichier.

La première étape d'ouverture du fichier en écriture est commune à tous les types de fichiers textuels. En supposant que le nom du fichier à ouvrir est stocké sous forme de chaîne de caractères dans la variable `nom_fichier`, le code suivant ouvre un fichier en écriture avec l'encodage UTF-8 et stocke dans la variable `fp` un pointeur sur l'endroit où nous sommes rendus dans notre écriture du fichier (pour l'instant, le début du fichier) :

```
fp = open(nom_fichier, "w", encoding="utf-8", newline="\n")
```

Le second argument (`"w"`) indique que le fichier doit être ouvert en mode *write*, donc en écriture.

Si le fichier en question existait déjà, son contenu est tout d'abord écrasé et on repart d'un fichier vide. Si l'on souhaite au contraire ajouter du texte à la fin d'un fichier existant, on utilisera le mode *append*, symbolisé par la lettre `"a"` :

```
fp = open(nom_fichier, "a", encoding="utf-8", newline="\n")
```

6.2.1 Fichiers textuels génériques

Pour ajouter du contenu à un fichier pointé par la variable `fp`, il suffit ensuite d'utiliser la méthode `write` :

```
fp.write("La vie est belle\n")
```

Notez que, contrairement à la fonction `print` à laquelle vous êtes habitué, la méthode `write` ne rajoute pas de caractère de fin de ligne après la chaîne de caractères passée en argument, il faut donc inclure ce caractère `"\n"` à la fin de la chaîne de caractères passée en argument, si vous souhaitez inclure un retour à la ligne.

6.2.2 Fichiers CSV

Le module `csv` déjà cité plus haut contient également une fonction `writer` permettant de simplifier l'écriture de fichiers CSV. La syntaxe d'utilisation de cette fonction est la suivante :

```
import csv

nom_fichier = "..." # À remplacer par le chemin vers le fichier :)

fp = open(nom_fichier, "w", encoding="utf-8", newline="\n")
csvfp = csv.writer(fp, delimiter=";"):
csvfp.writerow([1, 5, 7])
csvfp.writerow([2, 3])
# Après cela, le fichier contiendra les lignes suivantes :
# 1;5;7
# 2;3
```

La méthode `writerow` prend donc une liste en argument et écrit dans le fichier les éléments de cette liste, séparés par le délimiteur `";"` spécifié lors de l'appel à la fonction `writer`. Le retour à la ligne est écrit directement par la méthode `writerow`, vous n'avez pas à vous en occuper.

6.3 Manipulation de fichiers en Python avec le module `os`

Lorsque l'on lit ou écrit des fichiers, il est fréquent de vouloir répéter la même opération sur plusieurs fichiers, par exemple sur tous les fichiers avec l'extension `".txt"` d'un répertoire donné. Pour ce faire, on peut utiliser en Python le module `os` qui propose un certain nombre de fonctions standard de manipulation de fichiers. On utilisera notamment la fonction `listdir` de ce module qui permet de lister l'ensemble des fichiers et sous-répertoires contenus dans un répertoire donné :

```
import os

for nom_fichier in os.listdir("donnees"):
    print(nom_fichier)
```

La fonction `listdir` peut prendre indifféremment un chemin absolu ou relatif (dans notre exemple, il s'agit d'un chemin relatif qui pointe sur le sous-répertoire `"donnees"` contenu dans le répertoire de travail courant du programme).

Si vous exécutez le code ci-dessus et que votre répertoire `"donnees"` n'est pas vide, vous remarquerez que le nom du fichier stocké dans la variable `nom_fichier` ne contient pas le chemin vers ce fichier. Or, si l'on souhaite ensuite ouvrir ce fichier (que ce soit en lecture ou en écriture), il faudra bien spécifier ce chemin. Pour cela, on utilisera la syntaxe suivante :

```
import os

repertoire = "donnees"
for nom_fichier in os.listdir(repertoire):
    nom_complet_fichier = os.path.join(repertoire, nom_fichier)
    print(nom_fichier)
    print(nom_complet_fichier)
    fp = open(nom_complet_fichier, "r", encoding="utf-8")
    # [...]
```

La fonction `path.join` du module `os` permet d'obtenir le chemin complet vers le fichier à partir du nom du répertoire dans lequel il se trouve et du nom du fichier isolé. Il est préférable d'utiliser cette fonction plutôt que d'effectuer la concaténation des chaînes de caractères correspondantes car la forme des chemins complets dépend du système d'exploitation utilisé, ce que gère intelligemment `path.join`.

Exercice 6.1 Écrivez une fonction qui affiche, pour chaque fichier d'extension `".txt"` d'un répertoire passé en argument, affiche le nom du fichier ainsi que son nombre de lignes.

Chapitre 7

Tester son code

Dans ce document, nous avons jusqu'à présent supposé que tout se passait bien, que votre code ne retournait jamais d'erreur et qu'il ne contenait jamais de *bug*. Quel que soit votre niveau d'expertise en Python, ces deux hypothèses sont peu réalistes. Nous allons donc nous intéresser maintenant aux moyens de vérifier si votre code fait bien ce qu'on attend de lui et de mieux comprendre son comportement lorsque ce n'est pas le cas.

7.1 Les erreurs en Python

Étudions ce qu'il se passe lors de l'exécution du code suivant :

```
1 x = "12"
2 y = x + 2
```

Nous obtenons la sortie suivante :

```
Traceback (most recent call last):
  File "[...]", line 2, in <module>
    y = x + 2
TypeError: Can't convert 'int' object to str implicitly
```

Ce type de message d'erreur ne doit pas vous effrayer, il est là pour vous aider. Il vous fournit de précieuses informations :

1. l'erreur se produit à la ligne 2 de votre script Python ;
2. le problème est que Python ne peut pas convertir un objet de type `int` en chaîne de caractères (`str`) de manière implicite.

Reste à se demander pourquoi, dans le cas présent, Python voudrait transformer un entier en chaîne de caractères. Pour le comprendre, rendons-nous à la ligne 2

de notre script et décortiquons-la. Dans cette ligne (`y = x + 2`), deux opérations sont effectuées :

— la première consiste à effectuer l'opération + entre les opérandes `x` et `2` ;
— la seconde consiste à assigner le résultat de l'opération à la variable `y`.

Nous avons vu dans ce document que Python savait effectuer l'opération + avec des opérandes de types variés (nombre + nombre, liste + liste, chaîne de caractères + chaîne de caractères, et il en existe d'autres). Intéressons-nous ici au type des opérandes considérées. La variable `x` telle que définie à la ligne 1 est de type chaîne de caractères. La valeur `2` est de type entier. Il se trouve que Python n'a pas défini d'addition entre chaîne de caractères et entier et c'est pour cela que l'on obtient une erreur. Plus précisément, l'interpréteur Python nous dit : "si je pouvais convertir la valeur entière en chaîne de caractères à la volée, je pourrais faire l'opération + qui serait alors une concaténation, mais je ne me permets pas de le faire tant que vous ne l'avez pas écrit de manière explicite".

Maintenant que nous avons compris le sens de ce *bug*, il nous reste à le corriger. Si nous souhaitons faire la somme du nombre 10 (stocké sous forme de chaîne de caractères dans la variable `x`) et de la valeur 2, nous écrivons :

```
x = "12"
y = int(x) + 2
```

et l'addition s'effectue alors correctement entre deux valeurs numériques.

7.2 Les tests unitaires

Pour pouvoir être sûr du code que vous écrivez, il faut l'avoir testé sur un ensemble d'exemples qui vous semble refléter l'ensemble des cas de figures auxquels votre programme pourra être confronté. Or, cela représente un nombre de cas de figures très important dès lors que l'on commence à écrire des programmes un tant soit peu complexes. Ainsi, il est hautement recommandé de découper son code en fonctions de tailles raisonnables et qui puissent être testées indépendamment. Les tests associés à chacune de ces fonctions sont appelés **tests unitaires**.

Tout d'abord, en mettant en place de tels tests, vous pourrez détecter rapidement un éventuel *bug* dans votre code et ainsi gagner beaucoup de temps de développement. De plus, vous pourrez également vous assurer que les modifications ultérieures de votre code ne modifient pas son comportement pour les cas testés. En effet, lorsque l'on ajoute une fonctionnalité à un programme informatique, il faut avant toute choses s'assurer que celle-ci ne cassera pas le bon fonctionnement du programme dans les cas classiques d'utilisation pour lesquels il avait été à l'origine conçu.

Prenons maintenant un exemple concret. Supposons que l'on souhaite écrire une fonction `bissextile` capable de dire si une année est bissextile ou non. En se

renseignant sur le sujet, on apprend qu'une année est bissextile si :

— si elle est divisible par 4 et non divisible par 100, ou
— si elle est divisible par 400.

On en déduit un ensemble de tests adaptés :

```
print(bissextile(2004))  # True car divisible par 4 et non par 100
print(bissextile(1900))  # False car divisible par 100 et non par 400
print(bissextile(2000))  # True car divisible par 400
print(bissextile(1999))  # False car divisible ni par 4 ni par 100
```

On peut alors vérifier que le comportement de notre fonction `bissextile` est bien conforme à ce qui est attendu.

7.3 Le développement piloté par les tests

Le développement piloté par les tests (ou *Test-Driven Development*) est une technique de programmation qui consiste à rédiger les tests unitaires de votre programme avant même de rédiger le programme lui-même.

L'intérêt de cette façon de faire est qu'elle vous obligera à réfléchir aux différents cas d'utilisation d'une fonction avant de commencer à la coder. De plus, une fois ces différents cas identifiés, il est probable que la structure globale de la fonction à coder vous apparaisse plus clairement.

Si l'on reprend l'exemple de la fonction `bissextile` citée plus haut, on voit assez clairement qu'une fois que l'on a rédigé l'ensemble de tests, la fonction sera simple à coder et reprendra les différents cas considérés pour les tests:

```
def bissextile(annee):
    if annee % 4 == 0 and annee % 100 != 0:
        return True
    elif annee % 400 == 0:
        return True
    else:
        return False
```

Exercice 7.1 En utilisant les méthodes de développement préconisées dans ce chapitre, rédigez le code et les tests d'un programme permettant de déterminer le lendemain d'une date fournie sous la forme de trois entiers (jour, mois, année).

Chapitre 8

Conclusion

Dans ce document, nous avons abordé les principes de base de la programmation en Python, tels qu'enseignés à des étudiants non informaticiens de niveau Licence 2 (à l'Université de Rennes 2).

Comme indiqué en introduction, ce document se veut évolutif. N'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web.