Introduction à la programmation

Cours/travaux dirigés

SL25Y031

Timothée Bernard

Remarque et remerciements

Ce document constitue les notes de cours et les exercices de travaux dirigés du cours *Introduction à la programmation* enseigné au niveau L3 du parcours linguistique informatique [1] offert par l'UFR de linguistique de l'Université Paris Cité (code `SL25Y031`).

Ces notes s'inspirent fortement d'un cours du même nom construit de manière collective par l'UFR d'informatique de l'Université Paris Cité. En particulier, plusieurs exercices y sont ici repris avec peu voire pas de modification. Certaines explications ont aussi été reprises, bien que généralement adaptées. J'adresse donc un remerciement général à l'UFR d'informatique.

1. **Ceci** est un lien pointant vers la page web de cette formation.

Table des matières

Chapitre 1

Introduction

Objectifs :

- Comprendre la distinction entre expressions et valeurs.
- Comprendre ce que sont les types.
- Comprendre ce que sont les variables.
- Avoir une vision nette du concept d'association entre variables et valeurs.
- Intégrer le fait que les instructions d'un code doivent être écrites en suivant rigoureusement la syntaxe définie par PYTHON.
- Identifier et donner un sens aux différentes constructions du langage (affectation et utilisation des variables, structures conditionnelles, boucles *for* et fonctions).
- Être capable d'apporter une modification mineure à un programme existant.

1.1 Ce qu'est un programme

Programme et ordinateur [COURS]

- Un *programme* représente une séquence d'instructions à exécuter. Par exemple, une recette est un programme exécuté par un ou une cuisinière. Un code source PYTHON est un programme exécuté par un ordinateur. (Dans ce cours, nous utiliserons la version 3 de PYTHON.)
- Dans un cas très simple, un programme peut être directement une liste d'instructions à exécuter linéairement, de la première à la dernière. Dans le cas général, des *structures de contrôle* (appels de fonction, structures conditionnelles, boucles, etc.) permettent à la séquence d'instructions executées d'être beaucoup plus complexe que le code source lui-même.
- Remarque : Dans ce cours, l'objectif principal n'est pas d'apprendre à programmer dans l'« esprit PYTHON », mais d'apprendre à programmer clairement avec PYTHON. Cela signifie qu'en vue d'écrire du code plus lisible et plus facilement débuggable, nous nous nous écarterons parfois du « style PYTHON » souvent rencontré dans d'autres ressources.
- La plupart des choses que nous imaginons intuitivement à propos des ordinateurs sont des abstractions. La notion de bureau ou même l'arborescence du système de fichiers (la vision de la mémoire sous forme de fichiers localisés dans des dossiers potentiellement imbriqués) sont des abstractions. La mémoire est linéaire et uniforme : il s'agit simplement d'une séquence de bits (atomes de mémoire, pouvant valoir 0 ou 1) modifiables. Notons que la position d'un bit dans la séquence est son *adresse*. L'existence même de fichiers et de dossiers correspond à une interprétation de cette mémoire.
- Ces abstractions peuvent être très utiles pour conceptualiser l'ordinateur en vue d'un usage basique, mais certaines peuvent être problématiques (en nous induisant en erreur) lorsque l'on en fait un usage plus avancé.
- Il est très probable que certaines de vos conceptions conernant le fonctionnement des ordinateurs, de leur mémoire, des programmes, etc., soient inexactes et vous fournissent des intuitions inadaptées à la pratique de la programmation. Gardez cela en mémoire lorsque vous programmez : ce qui vous

paraît évident est peut-être en réalité faux. Repensez-y lorsque vous rencontrerez un problème qui vous paraît incompréhensible ; il faut que vous soyez prêt·e·s à remettre en question la plupart de vos conceptions sur les ordinateurs.
— Tout ce que fait un ordinateur se décompose en petites opérations élémentaires. Ces opérations sont extrêmement basiques et nous n'avons pas besoin de savoir de quoi il s'agit pour ce cours. Un langage de programmation comme PYTHON crée une couche d'abstraction au-dessus de ce niveau élémentaire, nous mettant à disposition des outils tels que la notion de type de valeur (entier, nombre à virgule, caractère, etc.), de variable ou de fonction ainsi que des opérations un petit peu plus complexes que les opérations élémentaires du processeur et avec lesquelles il est plus aisé de raisonner.
— Avec un langage de programmation moderne, il va être facile de lire des informations depuis un disque dur, le clavier ou internet, d'analyser et de transformer ces informations, puis d'envoyer le résultat vers un disque dur, l'écran ou internet.

Instruction et bloc de code [COURS]

— Un programme s'écrit principalement sous la forme d'une suite d'*instructions*. Une instruction décrit une action à effectuer et s'écrit généralement comme une ligne de code non vide.
— Parmi les instructions, les *affectations* consistent à assigner une *valeur* à une *variable*. En PYTHON (et comme dans de nombreux langages de programmation), on utilise l'opérateur = pour écrire des affectations. Il ne faut pas le confondre avec l'opérateur ==, que nous verrons un peu plus tard et qui est un opérateur de comparaison.
— La première des instructions suivantes est une affectation de la valeur 3 à la variable `x` et la seconde est une affectation de la valeur obtenue par l'évaluation de l'expression complexe `4 * x - 2` (c.-à-d. 10 si `x` vaut 3) à la variable `y` :

```
x = 3
y = 4 * x - 2
```

— Nous verrons qu'il existe bien d'autres types d'instructions que les affectations.
— Il est recommandé (voire demandé) d'inclure des *commentaires* dans son code. Les commentaires ne sont pas des instructions, ils ne sont pas exécutés ; ils servent principalement d'indications à l'attention des humains qui développent ou simplement lisent le code.
— Il existe plusieurs manières d'écrire un commentaire en PYTHON. Celle que nous allons utiliser dans ce cours fait commencer un commentaire par un croisillon (c.-à-d. le caractère `#`) et le fait finir à la fin de la ligne. Un commentaire peut être « seul » sur sa ligne, mais peut aussi être précédé d'une instruction :

```
# commentaire seul sur sa ligne
x = 3 # commentaire précédé d'une instruction
```

— Un bon code est, par définition, bien commenté. L'évaluation d'un devoir noté (quelle que soit la matière) comportant un exercice de programmation sera influencée par le code rendu et donc la qualité de ses commentaires. Nous verrons durant le semestre quelques bonnes pratiques concernant l'écriture des commentaires. L'idée principale est qu'ils doivent être clairs, précis et concis, et qu'ils doivent aider toute personne familière de la programmation à comprendre la logique du code au premier coup d'œil, ou presque.
— Une instruction ou une suite d'instructions peut former un *bloc de code* (ou « bloc d'instructions »). Certaines instructions — les structures de contrôle — peuvent déclencher ou non l'exécution de blocs de code. Par exemple, le code suivant contient deux blocs de code, un qui est tout simplement la totalité du code et un autre qui est contrôlé par une structure conditionnelle en `if` (et est inclus dans le premier) :

```python
# Début du premier bloc de code.
print('[BEGIN]') # Affiche '[BEING]' dans le terminal.

greeted = False # Affectation.

s = input() # Attend qu'une ligne de texte soit tappée au
    clavier et l'affecte à s.

if(s == 'greeting'): # Instruction conditionnelle.
  # Début du second bloc de code.
  print('Hi!')
  greeted = True # Réaffectation.
  # Fin du second bloc de code.

print(''greeted' is: ' + str(greeted))

print('[END]')
# Fin du premier bloc de code.
```

(Notons que ce code est — volontairement — trop commenté.)

— En PYTHON, les blocs de code se repèrent par leur *indentation* (présence d'espaces ou de tabulations en début de ligne) et non par les sauts de ligne. Comme dans l'exemple précédent, un bloc de code est souvent inclus dans un bloc de code plus grand (lui-même potentiellement inclus dans un bloc de code encore plus grand, etc.). Une instruction donnée peut donc appartenir à plusieurs blocs de code, mais il existe une notion de *plus petit bloc de code* auquel elle appartient.

— Le plus petit bloc de code auquel appartient une instruction s'obtient en partant de cette instruction et en incluant vers le haut toutes les instructions jusqu'à atteindre (i) une instruction d'indentation strictement plus faible ou (ii) une extrémité du code (en ignorant dans les deux cas les lignes vides), et de même vers le bas. Des instructions contiguës de même indentation font donc nécessairement partie des mêmes blocs de code.

Exercice 1 (Premier décodage, ⋆)
Dans le programme PYTHON *suivant, mettre en évidence les différents blocs de code puis classer les instructions en affectations, autres instructions et commentaires.*

```python
# start_hour, start_min, stop_hour, stop_min: int
def show_count_down(start_hour, start_min, stop_hour, stop_min):
  stop_secs = ((stop_hour * 60) + stop_min) * 60
  start_secs = ((start_hour * 60) + start_min) * 60
  number_of_secs = stop_secs - start_secs
  alert_code = 0
  for i in range(number_of_secs):
    time.sleep(1)

    if((number_of_secs - i) < 30):
      alert_code = 1
    print_int((number_of_secs - i), alert_code)
```

□

Importance de l'indentation [Cours]

— L'indentation du code en Python est cruciale ; elle fait partie de la syntaxe de Python.
— Si vous modifiez l'indentation d'un code correct, vous risquez très probablement (i) de le rendre ininterprétable ou (ii) de lui donner un tout autre sense.
— Par exemple, nous verrons que le code suivant :

```
x = 12
if(x > 0):
  print("La valeur associée à x est positive.")
  print(":)")
else:
  print("La valeur associée à x est négative.")
  print(":(")
```

produit l'affichage suivant :

```
1.  La valeur associée à x est positive.
2.  :)
```

Si l'on désindente la troisième ligne, alors l'exécution du code résultant échouera avec l'erreur suivante : « IndentationError : expected an indented block ». Si, à la place, l'on désindente la septième ligne, alors lexécution du code résultant produira l'affichage suivant :

```
1.  La valeur associée à x est positive.
2.  :)
3.  :(
```

1.2 Expressions, valeurs et types

Expressions et valeurs [Cours]

— Python est un langage et a donc une syntaxe. Tout comme la syntaxe du français définit différents types de constituants syntaxiques (ex : syntagme nominal, syntagme verbal, syntagme phrastique) et différentes manières de les combiner, la syntaxe du Python définit differents types de constituants syntaxiques et différentes manières de les combiner. Tout comme toute suite de mots en français ne forme pas forcément une phrase valide, toute suite de mots en Python ne forme pas forcément un programme valide.
— En Python, une catégorie syntaxique particulièrement interessante est celles des *expressions*. Une caractéristique sémantique des expressions est qu'elles sont *évaluables*, c.-à-d. que Python peut les traduire en valeur (ou « donnée »). D'une certaine manière, les expressions sont comparables a des syntagmes nominaux (ex : « le soleil », « Marie », « un vélo », « lui »), qui réfèrent à une entité.
— Une expression peut être syntaxiquement *simple* ou *complexe*.
— Les expressions simples sont les noms de constantes (ex : `2`, `-4.5`, `"Salut"`, `False`) et les noms de variables (ex : `x`, `word_count`). Les noms de constantes sont comparables à des noms propres et les noms de variables à des pronoms.
— Les expressions complexes sont construites à partir d'autres (sous-)expressions, elles-mêmes simples ou complexes, d'opérateurs et de symboles de ponctuation. Exemples : `1 + 3`, `(x * 2) > 7`, `print("Salut")`.
— De manière générale, la valeur d'une expression dépend du *contexte d'évaluation*, c.-à-d. des valeurs assignées aux différentes variables au moment de l'évaluation de l'expression. Il existe cependant des expressions dont l'évaluation est invariable ; c'est le cas des noms de constantes. (Cette situation est assez compatible avec la métaphore linguistique suggérée : « Barack Obama » réfère toujours à la même personne.)
— Pour connaître la valeur d'une expression, on peut généralement utiliser la fonction `print`. Par exemple, si `x` vaut 2, alors `print(x)` affiche « 2 ». (Notons que `print(x)` est elle-même une expression ; `print(print(x))` montre que sa valeur est `None`.)
— Un test simple pour vérifier qu'un fragment de code (non inclus dans un commentaire ou une chaîne de caractères) est une expression : On peut le mettre entre parenthèses et obtenir une instruction

syntaxiquement correcte. Par exemple, le code suivant s'exécute sans problème, ce qui confirme que `x`, `0`, mais aussi `x > 2`, sont des expressions :

```
x = 0 # Nécessaire pour que 'x' soit definie et que les
    instructions suivantes s'exécutent.
(x)
(0)
(x > 2)
```

Par contre, `x = 0` n'est pas une expression. En conséquence, si l'on rajoute des parenthèses autour de `x = 0` dans le code précédent, toute tentative d'exécution échouera, avec le message d'erreur suivant : « SyntaxError : invalid syntax ».

Types [Cours]

— En Python, toute valeur est d'un unique *type*. La fonction `type` permet de connaître le type d'une valeur. Par exemple :

```
type(3) # <class 'int'>
type(2 + 1) # <class 'int'>
x = 3; type(x) # <class 'int'>
type(0.5) # <class 'float'>
type(False) # <class 'bool'>
type(1 == 1) # <class 'bool'>
type(1 == 2) # <class 'bool'>
type('bonjour') # <class 'str'>
type([0, 1, 2]) # <class 'list'>
```

— Nous allons tout d'abord nous intéresser à deux *types numériques* : `int`, pour les *nombres entiers* (ou « entiers relatifs » ; ex : 0, 3, -1234), et `float`, pour les *nombres à virgule* (ex : 6.667, -0.001, 0.0, 3.0).

— On remarque qu'en Python, deux valeurs comme le `int` 3 et le `float` 3.0 peuvent être des représentations *distinctes* de ce que l'on considère intuitivement comme un *même* nombre. Il n'y a pas *identité* entre ces deux valeurs (`3 is not 3.0`) même s'il y a *égalité* (`3 == 3.0`).

— Le type est une propriété fondamentale de la valeur (autrement dit, deux valeurs ne peuvent pas être identiques si elles n'ont pas le même type). D'ailleurs (mais nous n'en dirons pas plus là-dessus), 3 et 3.0 ne sont pas du tout représentées (ni manipulées) de la même manière par l'ordinateur.

Manipulation des types numériques [Cours]

— Python met à notre disposition tout un ensemble d'*opérateurs arithmétiques* pour calculer des entiers à partir d'autres entiers.
  — Addition ; ex : `(7 + 3)` vaut 10.
  — Soustraction ; ex : `(7 - 3)` vaut 4.
  — Multiplication ; ex : `(7 * 3)` vaut 21.
  — Division entière ; ex : `(7 // 3)` vaut 2 (parce que $\mathbf{2} * 3 \leq 7 < (\mathbf{2} + 1) * 3$) et `(-7 // 3)` vaut -3 parce que $-\mathbf{3} * 3 \leq -7 < (-\mathbf{3} + 1) * 3$) ; la division entière « arrondit à l'inférieur », c.-à-d. que son résultat est égal au résultat de la division (non entière) arrondi à l'entier inférieur.
  — Modulo ; ex : `(7 % 3)` vaut 1, `(-7 % 3)` vaut 2 et, plus généralement, `(x % y)` est la valeur telle que `(x // y) * y + (x % y)` vaut x.
  — Puissance ; ex : `(7 ** 3)` vaut 343 (c.-à-d. $7 * 7 * 7$).
  Ces différentes opérations peuvent aussi être employées pour calculer des nombres à virgule à partir d'autres nombres à virgule (`(2.3 + 4.4)` vaut 6.7) ou d'un mélange d'entiers et de nombres à virgule (`(4 - 1.3)` vaut 2.7).

— Une barre oblique simple (`/`) désigne la division (non entière). Son résultat est toujours un `float`, que ses arguments soit des `float` ou des `int`. Par exemple, `(7 / 3)` vaut 2.3333333333333335 et `(6 / 3)` vaut 2.0.

— Il n'est pas toujours nécessaire d'utiliser autant de parenthèses car les expressions arithmétiques en

PYTHON suivent certaines conventions de notation standards. Les opérateurs `*`, `//`, `%`, `**` et `/` ont priorité haute alors que les opérateurs `+` et `-` ont priorité basse. Intuitivement, les opérateurs de priorité haute sont évalués avant les opérateurs de priorité basse et les opérateurs de même priorité sont évalués de la gauche vers la droite.

— Par exemple, `1 - 2 + 3` et `(1 - 2) + 3` s'évaluent de la même manière, et `1 - 2 * 3` et `1 - (2 * 3)` aussi. Cela dit, les parenthèses ne mangent pas de pain et peuvent grandement faciliter la vérification du code.

Exercice 2 (Python comme une calculatrice, ⋆)
Sans faire exécuter de code à la machine, donner la valeur des expressions suivantes :

```
6 * 7 + 3
6 * (7 + 3)
45 // 7
3 * 7 // 4
(3 * 7) // 4
(45 // 7) * 7 + 45 % 7
6 * -7 + 3
6 * (-7 + 3)
45 // -7
3 * -7 // 4
(3 * -7) // 4
(45 // -7) * -7 + 45 % -7
```

□

Exercice 3 (Évaluation et affichage, ⋆⋆)
Considérer les deux instructions suivantes :

```
3 + 4
print(3 + 4)
```

1. *La première de ces instructions est une expression ; que vaut-elle ?*
2. *Que produit l'exécution de la seconde de ces instructions ?*
3. *Cette seconde instruction est elle-même aussi une expression ; que vaut-elle ?*
4. *Que produit l'exécution de la première instruction ?*

□

1.3 Utilisation des variables

Affectation et réaffectation [COURS]

— Une *variable* a un *nom* et, durant l'exécution du programme et tant qu'elle est définie, est associée à une *valeur*.

— On assigne une valeur à une variable à l'aide de l'opérateur d'affectation, `=`. Par exemple, l'instruction suivante assigne à la variable `x` le résultat du calcul `6 * 7` :

```
x = 6 * 7
```

— Après exécution de l'instruction précédente, la valeur de la variable `x` est l'`int` 42.

— Alors que dans certains langages de programmation, il existe des instructions spécifiques devant être executées pour créer des variables avant de pouvoir leur affecter une valeur, en PYTHON, une variable est créée par la première affectation à celle-ci.

— Un nom de variable doit commencer par une lettre, qui peut être suivie d'autres lettres, de chiffres et de tirets bas (« _ »). Exemples : `x`, `y`, `x2`, `length`, `size_corpus`.

— D'un point de vue logique, les noms des variables n'ont pas d'importance (dans le sens où toutes les occurrences d'un nom de variable peuvent être remplacées par un même nouveau nom de variable sans affecter l'exécution du code), mais en pratique, bien choisir ses noms de variables est crucial pour faciliter la compréhension du code, sa lisibilité, son débuggage, etc.
— Comme mentionné dans la section sur les expressions, on peut utiliser la valeur d'une variable dans une expression en faisant référence à son nom. Ainsi, si `x` vaut 42, l'expression `(x + 1)` vaut 43.
— Utiliser dans une expression une variable qui n'a pas encore été créée n'a pas de sens. Si l'on essaye, lors de l'évaluation de l'expression, PYTHON *lève une exception* (≈ le programme plante).
— On peut réutiliser un nom de variable déjà utilisé dans une nouvelle affectation. Par exemple, après exécution du code suivant, `x` vaut 3 :

```
x = 12
x = 3
```

— Durant l'exécution d'un programme PYTHON, la machine maintient en mémoire les associations entre noms de variables et valeurs. On peut représenter visuellement ces associations en dessinant d'un côté un ensemble de noms de variables, de l'autre un ensemble de valeurs, et, pour chaque nom de variables, une flèche partant du nom de variable et pointant sur l'une des valeurs. (Il est possible qu'une valeur ne soit pointée par aucune flèche. Dans ce cas, la machine peut la supprimer de sa mémoire.)
— L'outil Python Tutor (https://pythontutor.com) permet d'exécuter du code Python instruction par instruction tout en représentant ces associations. Pour le programme suivant,

```
x = 3
y = 5
z = x
x = x + 3
```

voici ce que l'on peut voir :

1. `x` est un nom de variable non encore définie, l'exécution de `x = 3` ajoute un 3 du côté des valeurs, `x` du côté des variables, et une flèche de la variable vers la valeur ;
2. `y` est un nom de variable elle aussi non encore définie, l'exécution de `y = 5` ajoute un 5 du côté des valeurs, `y` du côté des variables, et une flèche de la variable vers la valeur ;
3. `z` est un nom de variable toujours non encore définie, l'exécution de `z = x` ajoute `z` du côté des variables et une flèche de celle-ci vers le 3 sur lequel pointe déjà `x` ;
4. l'exécution de `x = x + 12` ajoute un 15 (le résultat de `x + 12`) du côté des valeurs et redirige la flèche issue de `x` vers cette nouvelle valeur.

— Contrairement à d'autres langages de programmation, PYTHON accepte de réutiliser un nom de variable pour y affecter une valeur d'un type différent.

```
x = 12
print(type(x)) # Affiche "<class 'int'>".
x = 3.5
print(type(x)) # Affiche "<class 'float'>".
x = "vingt"
print(type(x)) # Affiche "<class 'str'>".
```

Bien que cela soit autorisé en PYTHON, en pratique, cela est généralement une mauvaise idée. Je vous déconseille d'utiliser le même nom de variable pour stocker des valeurs de type différent.

Exercice 4 (Nommer, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs. ***Pour ce genre d'exercices, lorsqu'une assignation est exécutée, commencer toujours par s'occuper de la valeur, puis de la variable, puis de la flèche.***

```
x = 3 * 5
y = x + x
```

□

Exercice 5 (Affectations, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
y = 4
# y = y + 1
x = y + 2
```

□

Exercice 6 (Affectations, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
y = x - 1
x = 2
```

□

Exercice 7 (Affectations, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
x = x - 1
```

□

Exercice 8 (Affectations, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
y = x
x = 3
```

□

Exercice 9 (Échange, ⋆⋆)
Étant données deux variables (définies) `x` *et* `y`*, écrire une suite d'instructions inversant les affectations de ces deux variables, c.-à-d. réassignant la valeur de* `x` *à* `y` *et inversement. Ne pas hésiter à introduire de nouvelles variables si nécessaire.* □

Variable et mémoire [Cours]

— On dit parfois à tort qu'une variable est une case mémoire contenant une valeur. *Cela est extrêmement trompeur.* Une variable doit plutôt être conçue comme une case mémoire contenant l'adresse d'une autre case mémoire contenant, elle, une valeur.
— Quand on affecte une valeur à un nom de variable, on stocke la valeur dans une case mémoire et on inscrit dans la variable l'adresse de cette case mémoire. Quand on affecte une nouvelle valeur à un nom de variable, on ne modifie pas la valeur stockée dans la première case mémoire, on crée (si besoin) une nouvelle case mémoire et on modifie l'adresse inscrite dans la variable.
— Cette conception correspond à la représentation des associations entre noms de variable et valeurs produite par Python Tutor.

1.4 Structures conditionnelles

Instruction conditionnelle [Cours]

— Une instruction conditionnelle permet de contrôler l'exécution d'un bloc de code en la faisant dépendre d'une *valeur booléenne* (c.-à-d. valant soit `True`, soit `False`).
— En Python, une structure conditionnelle se définit minimalement par :
  — une expression booléenne — la *condition* de la structure conditionnelle ;
  — un bloc de code — le *corps* de la structure conditionnelle.
— Une structure conditionnelle s'écrit à l'aide du mot-clef `if` suivi de la condition notée entre parenthèses et suivi d'un deux-points. Le corps est indiqué sur les lignes suivantes, indenté par rapport au mot-clef `if`.
— L'exécution du code suivant affiche « La condition est satisfaite. » et affecte la valeur de `x` à `y` à la condition que `x` vaille 0 ou moins.

```
if(x <= 0):
  print("La condition est satisfaite.")
  y = x
```

— Un corps composé d'une unique ligne peut être écrit directement après les deux-points de l'instruction conditionnelle plutôt que comme un bloc indenté. Par exemple, plutôt que d'écrire ce code-ci :

```
if(x <= 0):
  y = x
```

il est tout à fait possible d'écrire ce code-là :

```
if(x <= 0): y = x
```

— Lorsque l'on souhaite exécuter un bloc de code si une condition est vérifiée et un autre bloc de code si cette condition n'est pas vérifiée, plutôt que d'écrire deux structures conditionnelles avec des conditions contradictoires, il est possible d'utiliser le mot-clef `else`. Par exemple, plutôt que d'écrire ce code-ci :

```
if(x <= 0):
  y = x

if(x > 0):
  y = -x
```

il est fortement recommandé, pour des raisons de lisibilité, d'écrire ce code-là :

```
if(x <= 0):
  y = x
else:
  y = -x
```

Ces blocs de code équivalents permettent d'affecter la valeur de `x` à `y` si `x` vaut 0 ou moins et

d'affecter à y la valeur -x dans le cas contraire.

— Comme dans les exemples précédents, on construit très souvent les conditions des structures conditionnelles à l'aide d'un *opérateur de comparaison*.
  — Égalité : ==.
  — Inégalité : !=.
  — Infériorité stricte : <.
  — Infériorité large : <=.
  — Supériorité stricte : >.
  — Supériorité large : >=.
— Nous en apprendrons plus sur les structures conditionnelles ainsi que sur les valeurs booléennes dans les semaines suivantes.

Exercice 10 (Le max, ⋆)

1. *Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.*

```
x = 3
y = 4
if(x > y):
  xy_max = x
else:
  xy_max = y
```

2. *Transformer la suite d'instructions pour calculer le minimum de* x *et* y *et le mettre dans une variable* xy_min*.*

□

Exercice 11 (Différents tests, ⋆⋆)

Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
a = 2
b = a * a + 3
c = b - a

if(c == a):
  a = 1
else:
  a = a + 3

if(b + c < a):
  b = 2
else:
  b = 4

if(b != c * c):
  c = 12
else:
  c = -6
```

□

1.5 Boucles

Boucles *for* [Cours]

— Une boucle permet de répéter plusieurs fois le même bloc de code.
— Il existe en Python deux types de boucles : les boucles *for* et les boucles *while*. Les premières sont aussi appelées « boucles bornées » et les secondes « boucles non bornées ». Nous allons maintenant étudier les boucles *for* et étudierons les boucles *while* dans quelques semaines.
— La notion de boucle *for* repose sur la notion d'*itérable*. Un itérable en Python est un objet contenant des valeurs que l'on peut examiner les unes après les autres.
— Par exemple, en Python, une liste (type `list`) est un itérable dont les valeurs à examiner sont les éléments de la liste, et une chaîne de caractères (type `str`) est un itérable dont les valeurs à examiner sont les caractères de la chaîne.
— En Python, une boucle *for* se définit par :
  — un nom de variable — le *compteur* de la boucle ;
  — un itérable ;
  — un bloc de code — le *corps* de la boucle.
— Une boucle *for* s'écrit à l'aide du mot-clef `for` suivi du compteur, du mot clef `in`, de l'itérable et d'un deux-points. Le corps est indiqué sur les lignes suivantes, indenté par rapport au mot-clef `for`.
— La boucle suivante affiche les entiers de 0 *inclus* à 10 *exclu* :

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

— Notons que le compteur d'une boucle *for* n'est pas détruit à la fin de la boucle. Par exemple, le code suivant affiche tous les entiers de 0 inclus à 10 exclus, puis affiche à nouveau 9 — la valeur de `i` à la sortie de la boucle.

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

print(i) # Affiche '9'.
```

— Il est très rare qu'il soit utile d'accéder au compteur d'une boucle hors du corps de la boucle. D'ailleurs, dans beaucoup de langages de programmation (autres que Python), le compteur d'une boucle est détruit à la fin de son exécution. Il est une bonne pratique que d'éviter autant que possible — sauf éventuellement temporairement pour du débugage — d'accéder au compteur d'une boucle hors du corps de la boucle.
— Dans une boucle, c'est toujours le même bloc de code (le corps de la boucle) qui est exécuté à chaque itération. Cependant, le contexte d'évaluation de ce bloc, et en particulier la valeur du compteur, change. Ainsi, on ne répète pas forcément toujours exactement la même chose d'une itération sur l'autre. Dans l'exemple précédent, seule la valeur affichée (la valeur du compteur `i`) change d'une itération à l'autre ; nous verrons plus tard qu'en introduisant une structure conditionnelle dans le corps d'une boucle il est possible de faire varier considérablement les instructions exécutées lors des différentes itérations de la boucle.
— Beaucoup d'itérables courants (ex : les listes, les chaînes de caractères) associent une position bien précise à chacun des éléments concernés. Nous parlerons alors de *séquences* pour ces itérables à ordre déterminé.
— Certains itérables, par contre, n'associent pas de position particulière à leurs éléments. Par exemple, on ne peut pas savoir à l'avance dans quel ordre seront itérés les éléments d'un ensemble (type `set`) de taille deux ou plus. Dans le bloc de code suivant, `{"bonjour", "hello"}` est un ensemble contenant les deux chaînes de caractères `"bonjour"` et `"hello"`, et l'on ne peut pas savoir à l'avance laquelle des deux va être affichée en premier :

```
for s in {"bonjour", "hello"}:
  print(s)
```

La fonction *range* [Cours]

— N'importe quel itérable (en particulier une *liste*) permet de définir une boucle *for*. Nous verrons par exemple dans un prochain chapitre comment itérer sur les lignes d'un fichier texte, et nous concentrons maintenant sur certaines séquences d'entiers (c.-à-d. des itérables dont les éléments sont des entiers).

— Si `stop` est une valeur entière (type `int`), `range(stop)` est la séquence des entiers de 0 *inclus* à `stop` *exclu*, dans l'ordre croissant.

— Plus généralement, l'expression `range(start, stop, step)` définit une séquence telle que :
  — `start` est la valeur initiale (incluse) ;
  — `stop` est la valeur de sortie (exclue) ;
  — `step` est la différence entre deux valeurs successives.
  Autrement dit, `range(stop)` est équivalente à `range(0, stop, 1)`.

— Il est possible d'omettre seulement l'argument `step` : l'expression `range(start, stop)` est équivalente à `range(start, stop, 1)`.

— La fonction `print` appelée sur `range(stop)` n'indique pas explicitement quelles sont les différents éléments de cette séquence :

```
print(range(-2, 8, 3)) # Affiche "range(-2, 8, 3)".
```

Cependant, tout itérable peut être converti en une *liste* à l'aide de la fonction `list`, et la fonction `print` indique explicitement les différents éléments des listes :

```
print(list(range(-2, 8, 3))) # Affiche "[-2, 1, 4, 7]".
```

Cette fonctionnalité a notamment un intérêt pour tester/débugguer un programme en vérifiant qu'un itérable contient bien les valeurs attendues.

Exercice 12 (Afficher des suites d'entiers, ⋆)

1. *Écrire une boucle qui affiche les 100 premiers entiers, en commençant à 0.*
2. *Écrire une boucle qui affiche les entiers de -50 (inclus) à 49 (inclus).*
3. *Écrire une boucle qui affiche les 50 premiers entiers pairs, en commençant à 0.*
4. *Écrire une boucle qui affiche 1000 fois le nombre 3.*

□

Exercice 13 (Propriétés de la fonction *range*, ⋆)

1. *Soit deux entiers* `a` *et* `b` *tels que* `a <= b`. *Combien d'éléments contient* `range(a, b)` *?*
2. *Simplifier le code suivant :*

```
a = 12
b = 35
c = 72

for i in range(a, b):
  print(i)

for i in range(b, c):
  print(i)
```

□

1.6 Fonctions

Nommer un bloc de code [Cours]

— Python met à disposition un certain nombre d'*opérateurs* et de *fonctions natives* qui permettent d'écrire des instructions. Nous avons par exemple déjà vu l'opérateur d'affectation =, des opérateurs arithmétiques comme + et ∗, la fonction `type` ainsi que la fonction `print`. Nous découvrirons beaucoup d'autres fonctions natives, comme la fonction `input` qui permet d'obtenir une saisie au clavier.
— Les opérateurs et fonctions natives de Python sont suffisants pour écrire n'importe quel programme, cependant, quand une même opération complexe (faisant intervenir différents opérateurs ou fonctions natives) intervient plusieurs fois dans un programme, il est alors une bonne idée de définir une fonction (alors non native) qui implémente cette opération. Notons qu'il est courant de définir des fonctions même pour des opérations n'intervenant qu'une seule fois dans le code, pour des raisons de structuration et de lisibilité (et donc aussi de maintenance) du code.
— Une fonction se définit par :
  — un nom — le *nom* de la fonction ;
  — une série (éventuellement vide) de noms de variables — les *arguments* de la fonction ;
  — un bloc de code — le *corps* de la fonction.
— La définition d'une fonction commence par le mot-clef `def`, suivi du nom de la fonction, de ses arguments notés entre parenthèses et séparés par des virgules, puis d'un deux-points. Le corps de la fonction est indiqué sur les lignes suivantes, indenté par rapport au mot-clef `def`. Il est aussi courant de donner des informations sur la fonction en commentaire au-dessus de sa définition.
— Exemple de définition d'une fonction à deux arguments :

```
# x, y: int
def f(x, y):
  return ((x * 2) + (y * 3))
```

Exemple de définition d'une fonction à zéro argument :

```
def g():
  print("Bonjour. J'espère que vous allez bien.")
```

— Il est une très bonne idée de *systématiquement* indiquer en commentaire le·s type·s attendu·s pour chacun des arguments de toute nouvelle fonction définie.
— Une fois qu'une fonction est définie, on peut former des expressions à partir du nom de la fonction et d'une expression pour chaque argument, données entre parenthèses et séparées par des virgules. Une telle expression est un *appel de fonction*. Exemple :

```
# x: int
def hi_triple(x):
  print('Hi!')
  return (x * 3)

y = 4
z = hi_triple(y) + hi_triple(1 + 1) # Prints 'Hi!' twice and
    assigns 18 to `z`.
```

— Le corps d'une fonction peut contenir l'opérateur `return`. Cet opérateur sert à écrire un type d'instructions très particulières, les *instructions de retour*, qui permettent de définir la valeur des appels de fonction. L'opérateur `return` ne peut apparaître que dans le corps des fonctions.
— Quand un appel de fonction est évalué,
  — les expressions qui sont données pour les différents arguments sont d'abord évaluées,
  — les valeurs correspondantes sont affectées aux arguments de la fonction (des variables),
  — puis le corps de la fonction est évalué jusqu'à atteindre une instruction de retour ou la fin du corps de la fonction :
    — **dès qu'une instruction de retour est rencontrée, l'exécution de la fonction s'arrête**

par l'évaluation de l'expression indiquée après l'opérateur `return` et la valeur obtenue devient alors la valeur de l'appel de fonction,
— si aucune instruction de retour n'est rencontrée, la valeur de l'appel de fonction est `None`.

Exercice 14 (Utiliser une fonction, ⋆)
On considère la fonction suivante :

```
# x: int
def cube(x):
  return (x * x * x)
```

Après exécution de la suite d'instructions suivante, que vaut `a` *?*

```
b = 5
a = 3
a = cube(b - a)
```

□

Exercice 15 (Définir une fonction I, ⋆)
Considérer la définition de fonction suivante :

```
# x: float
def f(x):
  return ((x * x) + 5)
```

1. *Quel est le nom de la fonction définie ?*
2. *Réécrire la définition de manière à ce que la fonction retourne le résultat de la division entière de l'argument par 3.*
3. *Réécrire la définition de manière à ce que la fonction se nomme* `third`.

□

Exercice 16 (Définir une fonction II, ⋆⋆)
Définir des fonctions effectuant les calculs suivants et rétournant leurs résultats :

1. *Le produit de deux entiers donnés en argument, moins leur somme.*
2. *Le carré du produit de trois entiers donnés en argument.*

□

Exercice 17 (Fonctions récursives, ⋆⋆)

1. *Étant donnée la fonction* `f` *suivante, que vaut* `f(0)` *?* `f(1)` *?* `f(3)` *?* `f(0.5)` *?*

```
# n: int (>= 0)
def f(n):
  if(n == 0): return 0
  return f(n-1)
```

2. *Étant donnée la fonction* `g` *suivante, que vaut* `g(0)` *?* `g(1)` *?* `g(3)` *?* `g(0.5)` *?*

```
# n: int
def g(n):
  if(n <= 0): return 0
  return (1 + g(n-1))
```

□

Valeur de retour et affichage [Cours]

— Une fonction peut *afficher du texte à l'écran* à l'aide de la fonction `print`, comme dans l'exemple suivant :

```
# x, y: int
def f1(x, y):
  z = (3 * x) - (2 * y)
  print(z)
```

Une fonction peut *retourner une valeur* à l'aide d'une instruction de retour (avec le mot-clef `return`), comme dans l'exemple suivant :

```
# x, y: int
def f2(x, y):
  z = (3 * x) - (2 * y)
  return z
```

(Évidemment, une fonction peut commencer par afficher du texte et finir en renvoyant une valeur.)

— Afficher du texte à l'écran et renvoyer une valeur sont deux opérations *totalement distinctes*. Faites très attention aux énoncés des exercices. Quand il vous est demandé d'écrire une fonction, faites bien à attention à ce qu'elle doit afficher et à ce qu'elle doit retourner.

— Une valeur qui est simplement affichée ne va pas pouvoir être réutilisée, contrairement à une valeur retournée par une fonction. Il est courant d'affecter la valeur de retour d'une fonction à une variable pour la réutiliser par la suite :

```
a = f2(5, -1) # 17
b = 2 * a - 15 # La valeur de retour est réutilisée ici.
print(b) # Affiche "19".
```

— Les appels à la fonction `print` servent à la communication de la machine vers l'humain, alors que les instructions de retour servent à la communication au sein même de la machine (entre les différentes parties du programme).

— Comme indiqué plus haut, lors de l'évaluation d'un appel de fonction,
  — si une instruction de retour est exécutée, l'exécution s'arrête et la valeur de l'expression associée devient la valeur de l'appel de fonction,
  — si l'exécution de la fonction se termine sans rencontrer d'instruction de retour, la valeur de l'appel de fonction est `None`.

— `None` est une valeur de type `NoneType`. C'est la seule valeur de ce type. Cette valeur sert intuitivement à représenter l'« absence de valeur ».

— Comme l'exécution d'une instruction de retour stoppe net l'exécution de la fonction, la ou les instructions de retour ne doivent être exécutées que lorsque le calcul censé être effectué par la fonction est terminé.

— Il est important d'insister : *La différence entre affichage et valeur de retour est fondamentale*. Si vous avez le moindre doute à ce sujet, il faut le signaler.

Exercice 18 (Valeur de retour, ⋆⋆)

Pour chacune des suites d'instructions suivantes, décrire l'affichage produit pendant leur exécution et indiquer si l'exécution s'effectue sans erreur.

```
# x: int
def twice_a(x):
  return (2 * x)

y = twice_a(3)
print(y)
y = y + twice_a(1)
print(y)
```

```
# x: int
def twice_b(x):
  (2 * x)

y = twice_b(3)
print(y)
y = y + twice_b(1)
print(y)
```

2

```
# x: int
def twice_c(x):
  print(2 * x)

y = twice_c(3)
print(y)
y = y + twice_c(1)
print(y)
```

3

□

Exercice 19 (Affichage, ⋆)

1. *Quel est l'affichage produit par l'exécution de* `print(3)` *?*
2. *Quel est l'affichage produit par l'exécution de* `print(print(3))` *?*
3. *Quel est l'affichage produit par l'exécution de* `print(print(print(3)))` *?*

□

Exercice 20 (Évaluation d'une fonction, ⋆⋆)
Considérer la fonction `hi_triple` *définie comme suit :*

```
# x: int
def hi_triple(x):
  print('Hi!')
  return (x * 3)
```

1. *Que produit l'exécution de l'instruction suivante ?*

```
a = hi_triple(4)
```

2. *Que produit l'exécution de l'instruction suivante ?*

```
a = hi_triple(hi_triple(4))
```

□

Variables locales et arguments [Cours]

— Il est tout à fait possible de définir des variables dans le corps d'une fonction, par des instructions d'assignation. Ces variables, avec les arguments de la fonction, sont dites « locales (à la fonction) ». Les variables locales n'ont d'existence en tant que variable que pour la fonction.

— Durant l'exécution d'une fonction, une variable locale *masque* tout autre variable du même nom. Autrement dit, à l'intérieur d'une fonction, le nom d'une variable locale ne peut faire référence qu'à cette variable locale et non à une éventuelle variable de même nom déjà définie au moment de l'appel de fonction.

— Les variables locales d'une fonction disparaissent à la fin de l'exécution de la fonction. Les instructions de retour sont donc très importantes pour récupérer les valeurs calculées au sein d'une fonction et pouvoir y accéder même après la fin de son exécution.
— Pour le programme suivant,

```
# x: int
def f(x):
  y = x + 2
  return 3 * y

x = 12
z = f(x + 3)
```

voici ce que l'outil Python Tutor (https://pythontutor.com) montre :

1. `f` est initialement un nom de fonction non encore définie, l'exécution de la définition de fonction ajoute `f` du côté des variables, un `f(x)` du côté des valeurs pour représenter la fonction `f`, et une flèche de la première vers la seconde ;
2. `x` est un nom de variable non encore définie, l'exécution de `x = 12` ajoute `x` du côté des variables, un `12` du côté des valeurs, et une flèche de la première vers la seconde ;
3. au début de l'évaluation de l'appel de fonction `f(x + 3)`, une nouvelle `x` est ajouté du côté des variables pour représenter l'argument de `f` (une variable locale), un `15` (la valeur de `x + 3` au moment de l'appel de fonction) du côté des valeurs, et une flèche de la première vers la seconde ;
4. `y` est un nom de variable non encore définie localement dans `f`, l'exécution de `y = x + 2` ajoute `y` du côté des variables, un `17` du côté des valeurs, et une flèche de la première vers la seconde ;
5. l'exécution de l'instruction de retour ajoute « Return value » du côté des variables, un `51` du côté des valeurs, et une flèche de la première vers la seconde ;
6. `z` est un nom de variable non encore définie, l'exécution de `z = f(x + 3)` ajoute `z` du côté des variables et une flèche de celle-ci au `51` pointé par la « Return value » introduite par l'appel de fonction `f(x + 3)` ; de plus, les variables locales à `f` sont dorénavant grisées pour indiquer qu'elles ont disparues.

— Bien qu'il soit possible d'accéder dans une fonction à la valeur d'une variable non-locale, il existe à ce niveau des subtilités sur lesquelles nous reviendrons dans un prochain chapitre future (si le temps le permet). Pour le moment, si vous voulez utiliser la valeur d'une variable dans une fonction, passez-la lui comme argument.
— En pratique, la quasi-totalité des fonctions sont conçues pour recevoir des arguments d'un type (ou de quelques types) bien particulier(s). Cependant, par défaut, en PYTHON, les types des arguments ne sont pas contraints. Cela signifie que PYTHON ne vérifie pas si une valeur passée comme argument d'une fonction est bien d'un type adéquat. Lorsque ce n'est pas le cas, le comportement de la fonction est difficile à prévoir, mais son exécution risque de faire planter le programme. *Soyez donc très attentif·ve·s aux arguments que vous passez aux fonctions.*
— Petite astuce : Pour vérifier qu'une valeur `v` a bien le type attendu `t`, il est possible d'utiliser l'instruction `assert isinstance(v, t)`. L'expression `isintance(v, t)` vaut `True` si `v` est bien du type `t`, et `False` sinon ; l'instruction `assert isinstance(v, t)` n'a aucun effet si `v` est bien du type `t`, et arrête le programme en indiquant « AssertionError » sinon. Exemple :

```
# x: int
def triple(x):
  assert isinstance(x, int)

  return (x * 3)

y = triple(5) # Retourne 5.
z = triple("cinq") # Arrête le programme en indiquant "
    AssertionError".
```

Exercice 21 (Évaluation d'une fonction, ⋆⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
# x, y: int
def f(x, y):
  if(x < y):
    return (y - x)
  return ((2 * x) - y)

x = 5
y = 7
z1 = f(x, y)
z2 = f(y, x)
```

□

Exercice 22 (Fonction avec variable locale, ⋆⋆)

1. *En utilisant (de manière pertinente) une variable locale* `d`*, définir une fonction* `v_abs` *prenant en argument deux entiers* `m` *et* `n`*, et renvoyant la différence* `m - n` *si celle-ci et positive, et son opposé sinon.*
2. *Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs, et en détaillant l'affichage produit.*

```
d = 12
print(v_abs(7, 5))
print(d)
```

3. *Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs, et en détaillant l'affichage produit.*

```
print(v_abs(7, 5))
print(d)
```

□

1.7 Exercices supplémentaires

Exercice 23 (Valeurs d'expressions entières, ⋆)
Donner la valeur des expressions suivantes :

— `3 + 5 // 3`
— `4 * 1 // 4`
— `2 // 3 * 3 - 2`

□

Exercice 24 (Modulo 9, ⋆⋆)
Donner la valeur des expressions suivantes :

— `18 % 9`
— `81 + 18 % 9`
— `(81 + 18) % 9`

□

Exercice 25 (Valeur absolue, ⋆)
Définir une fonction retournant la valeur absolue d'une valeur de type `int` *donnée en argument (c.-à-d. la valeur elle-même si elle est positive et son opposée si elle est négative, de sorte que la valeur absolue est toujours positive). Le code ne doit pas faire appel à la fonction native* `abs` *mais être construit autour d'une structure conditionnelle.* □

Exercice 26 (Utiliser une fonction, ⋆)
On considère la fonction suivante :

```
# x, y: int
def sum_square(x, y):
  return (x * x + y * y)
```

Après exécution de la suite d'instructions suivante, que valent les différentes variables mentionnées ?

```
a = sum_square(2, 3)
b = sum_square(3, 1)
c = sum_square(4, 3)
```

□

Exercice 27 (Boucles simples, ⋆⋆)

1. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
for i in range(5):
  print(8)
```

2. *Après exécution de la suite d'instructions suivante, que vaut* `x` *?*

```
x = 0
for i in range(5):
  x = x + 8
```

3. *Après exécution de la suite d'instructions suivante, que vaut* `x` *?*

```
x = 0
for i in range(5):
  x = 10 * x + 8
```

□

Exercice 28 (Des entiers qui s'ajoutent, ⋆⋆)
Après exécution de la suite d'instructions suivante, que valent les différentes variables mentionnées ?

```
x = 0
i = 0
x = x + i
i = i + 1
x = x + i
i = i + 1
x = x + i
i = i + 1
x = x + i
i = i + 1
x = x + i
i = i + 1
```

Si l'on étendait le code ci-dessus pour faire non plus 5 mais 100 ou 1000 répétitions des deux instructions `x = x + i` *et* `i = i + 1`*, on obtiendrait un programme extrêmement long.*

On peut remplacer la répétition des deux instructions mentionnées par une boucle for en remarquant que l'instruction `x = x + i` *est exécutée 5 fois avec* `i` *prenant pour valeurs les différents entiers entre 0 et 4 car cette variable est systématiquement incrémentée (sa valeur est augmentée de 1) avec l'instruction* `i = i + 1`*.*

On obtient ainsi la boucle :

```
x = 0
i_lim = 5
for i in range(i_lim):
  x = x + i
```

1. *On remplace la seconde ligne du code précédent par* `i_lim = 100`*. Quelle est la valeur de* `x` *après l'exécution de cette suite d'instructions ?*
2. *Même question si l'on remplace la seconde ligne par* `i_lim = 1000`*.*
3. *Modifier le code précédent pour calculer la somme des entiers de 10 à 100.*
4. *Écrire une fonction* `sum_integers` *qui, pour tout entier* `n` *donné, renvoie la somme des entiers de 0 à* `n`*.*

□

Exercice 29 (Évaluation d'une fonction, ⋆⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
# x, y: int
def distance(x, y):
  if(x < y): return (y - x)
  return (x - y)

x = 63
y = 47
z1 = distance(x, y)
z2 = distance(y, x)
```

□

Exercice 30 (n-ième chiffre, ⋆⋆⋆)

Écrire une fonction prenant en argument un entier `k` *(supposé plus grand ou égal à 1) et un entier* `n` *(supposé plus grand ou égal à 0), et renvoyant :*

- *le* `k`*-ième chiffre de* `n` *en partant de la droite (dans l'écriture de* `n` *en base 10, c.-à-d. son écriture usuelle) si* `n` *s'écrit avec au moins* `k` *chiffres ;*
- *0 sinon.*

Par exemple, d'après cette définition, le premier chiffre de 1789 est 9, le second chiffre est 8, le troisième chiffre est 7, le quatrième chiffre est 1 et tous les autres sont 0.

Indice : Remarquer que `(1789 % 10)` *vaut 9,* `((1789 // 10) % 10)` *vaut 8,* `((1789 // 100) % 10)` *vaut 7, etc.* □

Chapitre 2

Expressions booléennes et structures conditionnelles

Objectifs :

- Maîtriser les expressions booléennes dans les conditions.
- Construire des structures conditionnelles complexes.

2.1 Les booléens

Expressions booléennes [Cours]

- Une expression booléenne s'évalue en une valeur de type `bool`. Il n'existe que deux valeurs de type `bool` : `True` et `False`.
- Une expression booléenne peut notamment être construite à partir de diverses expressions à l'aide des opérateurs de comparaison `==`, `!=`, `>`, `>=`, `<` et `<=`, ainsi qu'à partir d'autres expressions booléennes à l'aide des *connecteurs logiques*, qui sont `and`, `or` et `not`. Exemples d'expressions booléennes :
  - `5 != 4`
  - `(2 < x) and (x <= 5)`
  - `(x == 1) or (x == 2)`
  - `not (x == 0)` (qui est équivalent à `x != 0`)
- Les opérateurs de comparaisons peut s'utiliser de manière « chainée ». Par exemple, `(x < y < z)` est équivalente à `((x < y) and (y < z))`, et `(x > y < z)` est équivalente à `((x > y) and (y < z))`.
- Attention à ne pas confondre = et ==. Le premier est l'opérateur d'*affectation*, tandis que le deuxième est un opérateur de *comparaison*.
- La *conjonction* de deux expressions (construite avec `and`) est vraie ssi les deux expressions sont vraies.
- La *disjonction* de deux expressions (construite avec `or`) est vraie ssi l'une des deux expressions est vraie.
- La *négation* d'une expression (construite avec `not`) est vraie ssi cette expression n'est pas vraie.
- L'opérateur `not` est prioritaire sur l'opérateur `and` qui est lui-même prioritaire sur l'opérateur `or`. Pour être sûr·e de ne pas se tromper : mettre des parenthèses.

Exercice 31 (Savoir évaluer une condition, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
a = 2
a = a * a * a * a + 1
b = a // 2
c = a - 1
x = 0
if((b == 0) and (c == 16)):
  x = 1
else:
  x = 2
```

□

Exercice 32 (Évaluer, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 3 + 5
b = (x == (5 + 3))
d = (x > (x + 1))
e = (b or d)
f = (b and d)
f = (e != b)
```

□

Exercice 33 (Parité, ⋆)

1. *Écrire une fonction* `is_even` *prenant en argument un entier et renvoyant* `True` *si cet entier est pair et* `False` *sinon. La fonction doit être écrite autour d'une structure conditionnelle.*
   Indice : Utiliser l'opérateur `%` *(*modulo*).*
2. *Même question, mais sans structure conditionnelle.*

□

Équivalence entre expressions booléennes [Cours]

— On dit que deux expressions booléennes sont *équivalentes* quand ces deux expressions ont forcément la même valeur, quel que soit le contexte d'évaluation (c.-à-d. pour toutes les valeurs possibles des variables qui apparaissent dans ces expressions).
— Par exemple, en supposant que `x` a une valeur de type `int`, `((x > 10) and (x < 12))` et `(x == 11)` sont équivalentes. En effet, si `((x > 10) and (x < 12))` vaut `True`, alors `(x == 11)` faut forcément aussi `True` et inversement, si `(x == 11)` vaut `False`, alors `((x > 10) and (x < 12))` faut forcément aussi `False`.
— Par contre, `(x > y)` et `(x != y)` ne sont pas équivalentes : dans un contexte où `x=1` et `y=2`, par exemple, `(x > y)` vaut `False` alors que `(x != y)` vaut `True`.

Exercice 34 (Équivalence d'expressions, ⋆⋆)
Dire si les paires d'expressions booléennes suivantes sont équivalentes (en supposant que les variables mentionnées ont des valeurs de type `int`*) et, dans le cas où elles ne sont pas équivalentes, donner un exemple de contexte d'évaluation dans lequel les deux expressions ont des valeurs différentes :*

1. `((x > y) or (x < y))` *et* `(x != y)` ;
2. `((x != 3) and (x != 4) and (x != 5))` *et* `((x <= 2) or (x >= 6)` ;
3. `((x == y) and (x == z))` *et* `(x == z)` ;
4. `((x == y) and (x == z))` *et* `((x == y) and (y == z))`.

□

Exercice 35 (Simplification des expressions booléennes, ⋆⋆)
Simplifier *une expression booléenne consiste à la remplacer par une autre expression booléenne équivalente plus courte.*

Simplifier les expressions suivantes :
— `((x > 5) and (x > 7))`
— `((x == y) and (x == z) and (y == z))`
— `((x == 17) or ((x != 17) and (x == 42)))`
— `((x > 5) or ((x <= 5) and (y > 5)))`

□

Conversion en booléen [Cours]

— Il existe des règles de conversion des valeurs non booléennes vers les booléens. Ces règles permettent ainsi d'utiliser n'importe quelle valeur pour contrôler une structure conditionnelle. Par exemple, l'exécution du code suivant affiche « 'x' est non nul. » mais pas « 'y' est non nul. » car 0 est converti en `False` et tout autre entier (en particulier -3) en `True` :

```
x = -3
y = 0
if(x):
  print(''x' est non nul.')
if(y):
  print(''y' est non nul.')
```

— Le résultat de la conversion d'une valeur en booléen peut être connu avec la fonction `bool`. En particulier, si `x` est un entier, `bool(x)` vaut `False` si `x` vaut 0 et `True` sinon.
— Pour des raisons de lisibilité, je vous déconseille d'utiliser ce mécanisme de conversion implicite. Par exemple, le code précédent est équivalent au code suivant, plus explicite :

```
x = -3
y = 0
if(x != 0):
  print(''x' est non nul.')
if(y != 0):
  print(''y' est non nul.')
```

2.2 Retour sur les structures conditionnelles

Rappel sur les structures conditionnelles [Cours]

— Nous avons vu au premier chapitre que l'on pouvait contrôler l'exécution d'un ou de plusieurs blocs de code à l'aide d'une instruction conditionnelle. Exemple :

```
if(x <= 0):
  print("branche positive")
  y = x
else:
  print("branche négative")
  y = -x
```

— Nous avions vu que la branche négative (qui est exécutée quand la condition est fausse) était *optionnelle*. Exemple :

```
if(x <= 0):
  print("Erreur : x devrait être positif.")
```

Instructions imbriquées [Cours]

— Lorsqu'une instruction introduit un bloc de code, on dit que les instructions de ce bloc de code sont *imbriquées* dans la première instruction.

— Comme celle de n'importe quel langage de programmation standard, la grammaire de Python est *récursive* : une instruction peut être imbriquée dans une instruction de même type.

— Voici une instruction conditionnelle (directement) imbriquée dans une autre instruction conditionnelle :

```
x = 1
y = 2
z = 0
if(x == 1):
  if(y == 2):
    z = 3
  else:
    z = 5
else:
  z = 7
```

— Il n'y a pas de limite de profondeur à la récursion syntaxique, c.-à-d. que l'on a le droit d'imbriquer des instructions dans d'autres instructions à volonté : des conditionnelles dans des boucles dans des conditionnelles, etc. Pour des raisons de lisibilité, cependant, il est recommandé de limiter autant que possible la profondeur du code.

— Nous parlerons des boucles imbriquées au chapitre 3.

— Notons que le langage naturel aussi est récursif : un syntagme phrastique peut être imbriqué dans un autre (« Sabine sait que Jamy est journaliste »), un syntagme nominal peut être imbriqué dans un autre (« le chapeau de ma grand-mère »), etc.

Exercice 36 (Comprendre l'imbrication d'instructions, ⋆)
Expliquer la différence de comportement entre les trois fonctions suivantes en indiquant, pour chaque fonction, pour quelle·s valeur·s de `x` *le message « Au revoir. » est affiché :*

```
# x: int
def f(x):
  if(x > 0):
    print("‘x‘ est strictement positif.")
    if(x < 100):
      print("‘x‘ est strictement plus petit que 100.")
      print("Au revoir.")
```

```
# x: int
def g(x):
  if(x > 0):
    print("‘x‘ est strictement positif.")
    if(x < 100):
      print("‘x‘ est strictement plus petit que 100.")
    print("Au revoir.")
```

```
# x: int
def h(x):
  if(x > 0):
    print("‘x‘ est strictement positif.")
    if(x < 100):
      print("‘x‘ est strictement plus petit que 100.")
  print("Au revoir.")
```

□

Raccourci pour les conditionnelles en cascade [Cours]

— Lorsque la branche négative d'une structure conditionnelle commence immédiatement par une autre structure conditionnelle, il est possible et même recommandé d'utiliser le mot-clef `elif` pour limiter la profondeur du code et améliorer sa lisibilité. Par exemple :

```
x = 3
if(x == 0):
  print("‘x‘ est nul.")
else:
  if(x == 1):
    print("‘x‘ vaut un.")
  else:
    if(x == 2):
      print("‘x‘ vaut deux.")
    else:
      print("‘x‘ est strictement négatif ou strictement plus
   grand que deux.")
```

peut s'écrire :

```
x = 3
if(x == 0):
  print("‘x‘ est nul.")
elif(x == 1):
  print("‘x‘ vaut un.")
elif(x == 2):
  print("‘x‘ vaut deux.")
else:
  print("‘x‘ est strictement négatif ou strictement plus grand
    que deux.")
```

— Il est important de garder en tête que dans ces cas de conditionnels en cascade, une seule branche est exécutée : la première dont le test est satisfait. D'ailleurs, une fois qu'un test est satisfait, PYTHON n'évalue même pas les tests restants. Par exemple, dans le code suivant, la première condition étant satisfaite, seule la première branche, affichant « La variable est supérieure ou égale à 2 », est exécutée.

```
x = 2
if(x >= 2):
  print("La variable est supérieure ou égale à 2.")
elif(x <= 3):
  print("La variable est inférieure ou égale à 3.")
else:
  print("Autre cas.")
```

En particulier, la seconde branche n'est pas exécutée même si la seconde condition est satisfaite (ce que PYTHON ne vérifie même pas).

Exercice 37 (Utilisation des opérateurs booléens, ⋆⋆)
Remarque : Les fonctions données dans cet énoncé sont volontairement écrites de manière plus complexe que nécessaire et peuvent même contenir des instructions inutiles.

1. *D'après la définition de fonction suivante, que vaut* `f(y)` *pour chaque valeur de* `y` *prise dans* $\{0, 5, 15\}$ *?*

```
# y: int
def f(y):
  x = 0
  if(y != 0):
    if(y <= 10):
      x = 1
    else:
      x = 2
  else:
      x = 2

  return x
```

2. *Réécrire le corps de* `f` *en utilisant moins de tests* `if`*. Vérifier que la réécriture du code est correcte en vérifiant que les valeurs calculées pour* `f(y)` *sont toujours les mêmes.*

3. *D'après la définition de fonction suivante, que vaut* `g(a)` *pour chaque valeur de* `a` *prise dans* $\{0, 25, 50\}$ *?*

```
# a: int
def g(a):
  b = 2 * a
  b = a - a
  if(b == a):
    b = 0
  else:
    if(b > 43):
      b = 0
    else:
      b = 1

  return b
```

4. *Réécrire le corps de* `g` *en utilisant moins de tests* `if`*. Vérifier que la réécriture du code est correcte en vérifiant que les valeurs calculées pour* `g(a)` *sont toujours les mêmes.*
5. *D'après la définition de fonction suivante, que vaut* `h(c, d)` *pour chaque paire de valeurs pour le couple* (`c`, `d`) *prise dans* $\{(0,0),(0,6),(6,0),(6,6)\}$ *?*

```
# c, d: int
def h(c, d):
  e = 0
  if(d == 6):
    e = 3
  else:
    if(c >= 2):
      e = 2
    else:
      e = 3

  return e
```

6. *Réécrire le corps de* `h` *en utilisant moins de tests* `if`*. Vérifier que la réécriture du code est correcte en vérifiant que les valeurs calculées pour* `h(c, d)` *sont toujours les mêmes.*

□

Boucles et conditionnelles [Cours]

— Il est tout à fait possible de contrôler l'exécution de divers blocs de code au sein d'une boucle à l'aide d'une structure conditionnelle, notamment en fonction du compteur de la boucle. Par exemple, le programme suivant affiche « Pair. » lorsque `i` prend pour valeur un entier pair et « Impair. » lorsque `i` prend pour valeur un entier impair.

```
for i in range(100):
  if((i % 2) == 0):
    print("Pair.")
  else:
    print("Impair.")
```

Contrat Pour certaines questions, un *contrat* est indiqué. Il s'agit d'un ensemble de tests aidant à vérifier tout code proposé comme réponse. Si le code ne passe pas ces tests, c'est qu'il est érroné. (Si le code passe les tests du contrat, rien n'est garanti, mais c'est déjà ça.) Tout code proposé comme réponse à une question doit être testé ; quand aucun contrat n'est spécifié, vous devez vous-même trouver des tests.

Exercice 38 (Multiples de cinq, ⋆)
Écrire une fonction prenant en argument un entier n et affichant « Multiple de cinq. » pour les entiers entre 0 (inclus) et n (exclu) qui sont des multiples de cinq et « Pas multiple de cinq. » pour les autres.

Contrat :
Affichage pour `n=6` *:*

```
Multiple de cinq.
Pas multiple de cinq.
Pas multiple de cinq.
Pas multiple de cinq.
Pas multiple de cinq.
Multiple de cinq.
```

□

Exercice 39 (Multiples de cinq et de deux, ⋆⋆)

1. *Modifier minimalement la fonction de l'exercice précédent pour qu'elle affiche « Pair multiple de cinq. » pour les entiers entre 0 (inclus) et n (exclu) qui sont pairs et multiples de cinq, « Pair pas multiple de cinq. » pour ceux qui sont pairs mais pas multiples de cinq, et tout simplement « Impair. » pour les autres (les entiers impairs).*
2. *Simplifier la structure du code obtenu précédemment.*
   Astuce : Traiter le ou les cas les plus simples en premier.

□

Paramètre booléen [Cours]

— Il est très courant d'utiliser des arguments attendant des valeurs booléennes pour faire varier légèrement le comportement d'une fonction. Par exemple :

```
# n: int
# polite: bool
def buy_bread(n, polite):
  if(polite): print("Bonjour.")
  print("Je voudrais", n, "baguette(s)", end="") # 'end=""'
    demande à ne pas sauter à la ligne après l'affichage.
  if(polite): print(", s'il vous plaît.")
  else: print(".")

buy_bread(2, True) # Affiche "Bonjour." puis "Je voudrais 2
    baguette(s), s'il vous plaît.".
buy_bread(1, False) # Affiche "Je voudrais 1 baguette(s).".
```

Exercice 40 (Paramètre booléen, ⋆)
Écrire une fonction prenant en argument un entier `n` *et un booléen* `up`*, et affichant les entiers entre 1 et* `n` *(inclus), dans l'ordre croissant si* `up` *vaut* `True` *et dans l'ordre décroissant sinon.* □

2.3 Exercices supplémentaires

Exercice 41 (Condition et division entière, ⋆)
Soit un entier `x`*, écrire une expression booléenne indiquant si le tiers de* `x` *se trouve dans l'intervalle* $[2; 8]$.

Contrat :
x = 0 → *valeur : False*
x = 6 → *valeur : True*
x = 9 → *valeur : True*
x = 25 → *valeur : False*

□

Exercice 42 (Somme des cubes, ⋆)
Écrire à l'aide du boucle une séquence d'instructions calculant la somme des cubes des entiers de 1 à 100. □

Exercice 43 (Simplification de code, ⋆⋆)

1. *Dire, en justifiant, ce que fait la fonction suivante selon les valeurs entières données aux variables* `x` *et* `y`.

```
# x, y: int
def f(x, y):
  if(((x <= y) or (x >= y + 1))):
    if((x <= y) and (x < y)):
      if(x < -x):
        a = -x
        return a
      else:
        return x
    else:
      if(y < 0):
        return -y
  return y
```

2. *Écrire la même fonction de manière plus simple.*

□

Exercice 44 (Mention, ⋆)
Écrire une fonction `honour` *prenant en argument un entier* `grade` *et retournant la mention correspondante (une chaîne de caractères) : la mention est « passable » entre 10 inclus et 12 exclu ; « assez bien » entre 12 inclus et 14 exclu ; « bien » entre 14 inclus et 16 exclu ; « très bien » au-delà de 16 et non définie (*`None`*) sinon.*

Contrat :
`grade = 12` → *retour :* `"assez bien"`
`grade = 3` → *retour :* `None`

□

Exercice 45 (Nombres parfaits, ⋆⋆⋆)
Un entier $n > 1$ *est dit* parfait *s'il est égal à la somme de ses diviseurs propres (c.-à-d. autres que lui-même). Par exemple, 6 est parfait car ses diviseurs propres sont 1, 2 et 3, et on a* $6 = 1 + 2 + 3$.

1. *Écrire une fonction* `divisors` *qui prend en argument un entier* `n` *et affiche tous les entiers strictement inférieurs à* `n` *qui divisent* `n`.

Contrat :
n = 1 → *affichage :*
n = 4 → *affichage : 1 2*
n = 5 → *affichage : 1*
n = 60 → *affichage : 1 2 3 4 5 6 10 12 15 20 30*

2. *Écrire une fonction* `is_perfect` *qui prend en argument un entier* `n` *et qui affiche « n est parfait. » (où n est remplacé par la valeur de* `n`*) si* `n` *est parfait.*
3. *Écrire une fonction* `enum_perfect` *qui prend en argument un entier* `m` *et qui, pour tout nombre parfait* `n` *inférieur ou égal à* `m`*, affiche « n est parfait. » (où n est remplacé par la valeur de* `n`*).*

□

Chapitre 3

Chaînes de caractères

Objectifs :

- Effectuer des manipulations complexes des chaînes de caractères.
- Construire des chaînes de caractères par concaténation et à l'aide de chaînes de caractères formatées.
- Savoir utiliser les chaînes de caractères formatées (f-strings).
- Récupérer une saisie de texte depuis le clavier.

3.1 Les chaînes de caractères

Le type `str` [Cours]

- Le type `str` est le type des *chaînes de caractères*, qui sont conçues pour représenter du texte.
- Les chaînes de caractères sont écrites entre guillemets doubles (`"`) ou, de manière équivalente, entre guillements simples (`'`). Par exemple, `"M. Hulot"` désigne une chaîne de caractères et `'M. Hulot'` désigne exactement la même chaîne.
- Pour introduire un guillemet double dans une chaîne notée entre guillemets doubles, on écrit `\"` . Idem pour un guillemet simple dans une chaîne notée entre guillemets simples. On dit d'un tel caractère qu'il est *échappé*. Par exemple, `print("L'acronyme du nom de mon université est \"UPC\".")` provoque l'affichage de « L'acronyme du nom de mon université est "UPC". ».
- `"\"UPC\""` désigne la même chaîne que `'"UPC"'`.
- Bien que ce ne soit pas toujours indispensable, il est en général une très bonne idée (parfois nécessaire) d'échapper aussi les antislashs (\). Par exemple, `print("Rosny\\Bois")` provoque l'affichage de « Rosny\Bois ».
- La longueur d'une chaîne de caractères s'obtient grâce à la fonction `len`. Par exemple :

```
s_len = len("Hello!")
print(s_len) # Affiche "6".
```

- L'unique chaîne de caractères de longueur 0 est la chaîne vide, notée aussi bien par `""`, `''` ou `str()`.
- Dans une chaîne, « \n » représente le passage à une nouvelle ligne. Il s'agit en Python d'un unique caractère (`len("\n")` vaut 1), qui n'est pas affiché comme un symbole mais produit un retour à la ligne.
- On peut utiliser les comparateurs usuels sur les chaînes de caractères. Ainsi, == permet de tester si deux chaînes de caractères sont égales et != si deux chaînes sont différentes. Par exemple :

```
# name: str
def profiler(name):
  if((name == "Paul") or (name == "Pierre")):
    print("On a trouvé Paul ou Pierre.")
  else:
    print("Ce n'est ni Paul, ni Pierre.")
```

— L'ordre > sur les chaînes de caractères est une variante de l'ordre lexicographique (c.-à-d. celui du dictionnaire ; « plus grand » voulant dire « après dans le dictionnaire »). Cet ordre place les caractères accentués après leurs versions non accentuées et les caractères en bas de casse (les minuscules) après leurs versions en haut de casse (les majuscules). Par exemple :

```
print("zèbre" > "alpha") # Affiche "True".
print("alpha" > "al") # Affiche "True".
print("à" > "a") # Affiche "True".
print("a" > "A") # Affiche "True".
```

— Le mot-clef `in` permet de construire des expressions booléennes dénotant si une chaîne est contenue (de manière jointe) dans une autre chaîne. Par exemple, `("soif" in "assoiffée")` et `("a" in "assoiffée")` valent `True` alors que `("z" in "assoiffée")` et `("asife" in "assoiffée")` valent `False`. On peut noter que pour toute chaîne de caractères `s`, `("" in s)` vaut `True`.

— Les méthodes `startswith` et `endswith` permettent de savoir si une chaîne, respectivement, commence ou se termine par une autre chaîne. Par exemple, `"enfin".startswith("en")` et `"pourquoi".endswith("quoi")` valent `True`.

Exercice 46 (Début et fin de chaîne, ⋆)
Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?

```
word = "Déconstruire"
if(word.startswith("dé")): print("coucou 1")
if(word.startswith("Dé")): print("coucou 2")
if(word.endswith("Dé")): print("coucou 3")
if(word.endswith("uire")): print("coucou 4")
```

□

Exercice 47 (Espace et saut de ligne, ⋆)
Écrire une fonction `f` *prenant en argument une chaîne de caractères* `s` *et retournant* `True` *si* `s` *contient au moins un espace ou un saut de ligne, et* `False` *sinon.* □

Exercice 48 (Guillemet simple ou double, ⋆)
Écrire une fonction `f` *prenant en argument une chaîne de caractères* `s` *et retournant* `True` *si* `s` *contient au moins un guillemet (simple ou double), et* `False` *sinon.* □

Exercice 49 (Parité de longueur, ⋆⋆)
Écrire une fonction `f` *prenant en argument une chaîne de caractères* `s` *et affichant* `"La longueur est paire."` *si* `s` *est de longueur paire, et* `"La longueur est impaire."` *sinon.* □

Quelques transformations de chaînes [Cours]

— L'opérateur de concaténation de deux chaînes de caractères est +. Par exemple, `("vin" + "aigre")` vaut `"vinaigre"`.

— L'opérateur `*` appliqué à une chaîne `s` et un entier `n` produit la chaîne obtenue en répétant `s` `n` fois. Par exemple, `("-- " * 4)` vaut `"-- -- -- -- "`.

— Les méthodes `upper` et `lower` permettent d'accéder à la chaîne obtenue en prenant la version en, respectivement, haut de casse ou bas de casse de chaque caractère d'une chaîne. Par exemple, `"Que fait 50 Cent ?".upper()` vaut `"QUE FAIT 50 CENT ?"` et `"Que fait 50 Cent ?".lower()` vaut `"que fait 50 cent ?"`.

— La méthode `replace` appelée sur une chaîne `s` avec comme arguments deux chaînes `old_s` et `new_s` retourne la chaîne obtenue de `s` en y substituant toute occurrence de `old_s` par une occurrence de `new_s`. Par exemple :

```
s = "tata"
print(s.replace("a", "on")) # Affiche "tonton".
print(s.replace("ta", "pom")) # Affiche "pompom".
print(s) # Affiche "tata".
```

— Notons que la concaténation, la répétition, `upper`, `lower` et `replace` génèrent de nouvelles chaînes sans altérer les chaînes de départ (on dit que ce sont des opérations *fonctionnelles*). Ce fait est notamment illustré pour `replace` dans l'exemple précédent (c'est bien `"tata"` qui est affichée lors de l'exécution de la dernière instruction).

Exercice 50 (Concaténations de chaînes, ⋆)
Après exécution de la suite d'instructions suivante, que valent les différentes variables mentionnées ?

```
s = " rentre chez lui."
s2 = "Paul"
s3 = "Michel"
s4 = s2 + s
s5 = s3 + s
s6 = s2 + " et " + s3 + " rentrent chez eux."
```

□

Exercice 51 (Changement de casse, ⋆)
Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?

```
word = "Déconstruire"
if(word.lower().startswith("Dé")): print("coucou 1")
if(word.lower().startswith("Dé".lower())): print("coucou 2")
if(word.upper().endswith("IRE")): print("coucou 3")
if(word.upper().endswith("")): print("coucou 4")
if("CoNsT".lower() in word.upper()): print("coucou 5")
if("CoNsT".lower() in word.lower()): print("coucou 6")
```

□

Exercice 52 (Manipulation trompeuse de chaînes, ⋆⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
# s: str
def f(s):
  s = s + s
  s.upper()

s1 = "Bonjour."
s1.lower()
s2 = f(s1)
```

□

Exercice 53 (Plus aucun espace, ⋆)
Écrire une fonction `withoutSpace` *prenant en argument une chaîne de caractères* `s` *et retournant la chaîne obtenue de* `s` *en supprimant tous les espaces.*

Contrat :
`s` = `"cher M. Patate"` → *retour :* `"cherM.Patate"`

□

Calcul, affichage et valeur de retour [Cours]

— Cela peut paraître évident, mais il est important de garder en tête que lorsque l'on exécute un script Python, les valeurs, en particulier les chaînes de caractères, ne sont pas automatiquement affichées lorsqu'elles sont calculées. Par exemple, le code suivant ne produit aucune sortie et se contente d'assigner les chaînes `"Marguerite Duras"`, `"1914"` et `"Marguerite Duras est née en 1914."` aux variables `name`, `year` et `sentence` respectivement :

```
name = "Marguerite Duras"
year = "1914"
sentence = name + " est née en " + year + "."
```

Dans ce code, ces variables disparaissent à la fin de l'exécution sans avoir vraiment servi à rien.

— Si l'on souhaite afficher la chaîne produite par concaténation dans le code précédent, alors il faut introduire un appel à `print` :

```
name = "Marguerite Duras"
year = "1914"
sentence = name + " est née en " + year + "."
print(sentence)
```

(On peut aussi se passer de la variable `sentence` et écrire directement `print(name + " est née en " + year + ".")`.)

— Au passage, rappelons que par défaut, la valeur de retour d'une fonction est toujours `None` et que toute autre valeur doit impérativement être explicitement indiquée par une instruction de retour. C'est pourquoi les variables `x1`, `x2` et `x3` valent toutes trois `None` après exécution du code suivant :

```
# name, year: str
def f1(name, year):
  sentence = name + " est née en " + year + "."

x1 = f1("Marguerite Yourcenar", "1914") # None

# name, year: str
def f2(name, year):
  name + " est née en " + year + "."

x2 = f2("Marguerite Yourcenar", "1914") # None

# name, year: str
def f3(name, year):
  print(name + " est née en " + year + ".")

x3 = f3("Marguerite Yourcenar", "1914") # None
```

— Rappelons aussi que la valeur de retour de la fonction `print` est toujours `None`. Ainsi, dans le code suivant, les deux appels à `f4` et `f5` produisent bien chacun un affichage de la chaîne calculée, mais seul l'appel à `f5` renvoie cette chaîne :

```
# name, year: str
def f4(name, year):
  return print(name + " est née en " + year + ".")

x4 = f4("Marguerite Yourcenar", "1914") # None

# name, year: str
def f5(name, year):
  sentence = name + " est née en " + year + "."
  print(sentence)

  return sentence

x5 = f5("Marguerite Yourcenar", "1914") # "Marguerite
    Yourcenar est née en 1914."
```

Exercice 54 (Affichage et valeur de retour, ⋆)

1. *On cherche à écrire une fonction prenant une* `str` *en argument et affichant un message différent suivant s'il s'agit du nom de l'un des Rois mages (c.-à-d. « Melchior », « Gaspar » ou « Balthazar ») ou non. Considérer la proposition suivante :*

```
# name: str
def isMagi(name):
  if((name == "Melchior") and (name == "Gaspard") and (name ==
   "Balthazar")):
    res = "Il s'agit de l'un des Rois mages."
  else:
    res = "Il ne s'agit pas de l'un des Rois mages."
```

   (a) *Expliquer pourquoi cette proposition ne convient pas (il peut y avoir plusieurs problèmes).*
   (b) *Effectuer une correction minimale de cette proposition.*
2. *On souhaite maintenant que la fonction retourne le message (plutôt que de l'afficher). Effectuer à cette fin une correction minimale de la proposition précédante.*
3. *On souhaite maintenant que la fonction retourne simplement un booléen indiquant si le nom donné en argument est celui de l'un des Rois mages. Effectuer à cette fin une correction minimale de la proposition précédante.*

□

3.2 Utilisation des différents types `int` et `str`

Les chaînes de caractères et les entiers sont de type différent [Cours]

— L'opérateur + dénote différentes fonctions dépendamment des arguments avec lesquels il est utilisé. Par exemple, si appliqué à deux entiers, il dénote l'addition, et si appliqué à deux chaînes, il dénote la concaténation. Par contre, il ne dénote aucune fonction permettant d'associer un entier et une chaîne de caractères. Ainsi, la fonction add définit dans le code suivant s'exécute sans problème sur un argument de type `int` mais génère une *erreur de typage* lors de son exécution sur un argument de type `str` :

```
# x: int
def add(x):
  return (x + 2)

print(add(4)) # Affiche "6".
print(add("4")) # Plante.
```

— Certaines fonctions acceptent autant les `int` que les `str`. C'est le cas par exemple de `print`, qui accepte d'ailleurs n'importe quel type de valeur.
— On peut convertir une donnée de type `int` en chaîne de caractères grâce à fonction `str`. Par exemple :

```
s1 = str(17) # "17"
s2 = "J'ai " + s1 + " ans." # "J'ai 17 ans."
age = 33
s3 = "J'ai " + str(age) + " ans." # "J'ai 33 ans."
```

— Inversement, on peut convertir une chaîne de caractères représentant un entier en la valeur de type `int` correspondante grâce à la fonction `int`. Par exemple, `int("32")` vaut l'`int` 32.
— La fonction `int` génère une erreur si l'argument qui lui est passé n'est pas une représentation standard d'un entier. Par exemple, l'exécution de `int("deux")`, `int("3.5")` ou `int("zorglub")` génère une erreur.

— Similairement, la fonction `str` permet de convertir tout `float` (nombre à virgule) en une représentation textuelle que la fonction `float` permet de convertir en valeur de type `float` :

```
x = float("4.6") # 4.6
y = str(x) # "4.6"
```

— Attention, la fonction `bool` évoquée précédemment ne fonctionne pas de la même manière, dans le sens où `bool("False")` ne vaut pas `False` ! (`bool` renvoie `False` pour la chaîne vide et `True` pour toute autre chaîne.)

Exercice 55 (Deux représentations très différentes de la même chose, ⋆⋆)

1. *Que vaut l'expression* `("41" + "1")` *?*
2. *Que vaut l'expression* `(41 + 1)` *?*
3. *Quelle différence faites-vous entre la* `str "41"` *et l'*`int 41` *? Dans quelles situations utiliser l'une ou l'autre de ces représentations du nombre 41 ?*

□

Exercice 56 (Trouver le bon type, ⋆⋆)
Pour chacune des fonctions suivantes, préciser le types de valeurs autorisés pour les arguments et le type des valeurs retournées. On ne considérera comme types ici que `int` *et* `str`*.*

1.
```
# x: ?
def f(x):
  y = x % 2
  if(y == 0):
    return "Argument pair"
  else:
    return "Argument impair"
```

2.
```
# x: ?
def g(x):
  y = x % 2
  if(y == 0):
    print("Argument pair")
  else:
    print("Argument impair")
  return y
```

3.
```
# x: ?
# y: ?
def h(x, y):
  return (x + y)
```

4.
```
# x: ?
def id(x):
  return x
```

□

Exercice 57 (Conversion en booléen, ⋆⋆)
Définir une fonction `to_bool` *qui soit la réciproque de* `str` *pour les deux valeurs booléennes, c.-à-d. telle que* `to_bool(str(True))` *vaille* `True` *et* `to_bool(str(False))` *vaille* `False`*.* □

Chaînes de caractères formatées (f-strings) [Cours]

— Il existe (depuis Python 3.6) une syntaxe particulière très pratique pour créer des `str` : les chaînes de caractères formatées (en anglais : *formatted strings* ou *f-strings*).

— Une chaîne de caractères formatée s'écrit comme une constante de chaîne de caractères préfixée du symbole « f ». Par exemple : `f"une f-string"`, `f'Je suis {name}.'`, `f"2 * 1 + 3 = {2 * 1 + 3}"`.

— La valeur d'une chaîne de caractères formatée, de type `str`, est obtenue en remplaçant toute sous-expression entre accolades par sa valeur après conversion en `str`.

— Par exemple, les deux codes suivants sont équivalents :

```
name = "Agatha Christie"
print(f"Je suis {name}.") # Affiche "Je suis Agatha Christie
    .".

print(f"2 * 1 + 3 = {2 * 1 + 3}") # Affiche "2 * 1 + 3 = 5".

start = 12
stop = 203
step = 3
sentence = f"La séquence des entiers de {start} (inclus) à {
    stop} (exclu) par pas de {step} se construit avec 'range({
    start}, {stop}, {step})'."
```

```
name = "Agatha Christie"
print("Je suis " + name + ".") # Affiche "Je suis Agatha
    Christie.".
print("2 * 1 + 3 = " + str(2 * 1 + 3)) # Affiche "2 * 1 + 3 =
    5".

start = 12
stop = 203
step = 3
sentence = "La séquence des entiers de " + str(start) + " (
    inclus) à " + str(stop) + " (exclu) par pas de " + str(step
    ) + " se construit avec 'range(" + str(start) + ", " + str(
    stop) + ", " + str(step) + ")'."
```

Exercice 58 (Chaînes de caractères et chaînes de caractères formatées, ⋆⋆)

Après exécution de la suite d'instructions suivante, que valent les variables `s1`, `s2` *et* `s3` *(préciser le type de chacune de ces trois valeurs) ?*

```
x = 12
y = 7
s1 = f"If {x} > {y}, then {x} is larger that {y}."
s2 = "If {x} > {y}, then {x} is larger that {y}."
s3 = f"If {x > y}, then {x} is larger that {y}."
```

□

Exercice 59 (Construire une chaîne de caractères formatée, ⋆⋆)

1. *En utilisant une chaîne de caractères formatée, définir une fonction* `f` *prenant en argument deux* `int` `x` *et* `y`*, et renvoyant la chaîne de forme « a x b = c » où « a », « b » et « c » sont respectivement remplacés par* `x`*,* `y` *et leur produit.*

   Contrat :

   `x, y` = 3, 4 → *retour :* `"3 x 4 = 12"`

2. *Réécrire cette fonction sans utiliser de chaînes de caractères formatées.*

□

3.3 Manipulation du type `str`

Parcourir une chaîne de caractères [Cours]

— Il est possible d'accéder aux différents caractères d'une chaîne de caractères à partir de leur position dans la chaîne.

— Les caractères d'une chaîne `s` de caractères de longueur l sont indicés de 0 pour le premier à $l-1$ pour le dernier.

— L'on accède au caractère de position `i` d'une chaîne `s` avec l'expression `s[i]`. Par exemple, si l'on a `s="Salut"`, `s[0]` vaut `"S"` — le caractère en position 0 de la chaîne (le premier). De même, `s[1]` vaut `"a"`, `s[2]` vaut `"l"`, etc.

— Notons qu'en Python, un caractère est une chaîne de caractères de longueur 1 ; à l'inverse d'autres langages de programmation, il n'y a pas de type spécifique pour les caractères en Python.

```
print("Salut"[0]) # Affiche "S".
print(type("Salut"[0])) # Affiche "<class 'str'>".
print(type("Salut"[0]) == type("Salut")) # Affiche "True".
```

Cela implique notamment que tout ce que l'on peut faire avec une chaîne de caractères peut être fait avec un caractère. Par exemple, `"Salut"[0].lower()` est une expression correcte et vaut `"s"`.

— Si l'on cherche à accéder dans une chaîne de caractères à une position supérieure ou égale à la longueur de la chaîne, une erreur sera produite à l'exécution.

— Les entiers négatifs de `-1` à `-len(s)` (inclus) indicent aussi les caractères de `s`, du dernier au premier. Ainsi, `s[-1]` a la même valeur que `s[len(s)-1]` — le dernier caractère de `s` —, et `s[-len(s)]` a la même valeur que `s[0]` — le premier caractère de `s`.

— Si l'on cherche à accéder dans une chaîne de caractères à un indice strictement inférieur à l'opposé de la longueur de la chaîne, une erreur sera produite à l'exécution.

— Une chaîne de caractères est un itérable, on peut donc s'en servir pour écrire une boucle *for*. Les éléments d'une chaîne de caractères sont justement les caractères, qui sont ordonnés de la gauche vers la droite ; il s'agit donc de ce que nous avons appelé une séquence. Par exemple, le code suivant affiche chaque caractère d'une chaîne sur une ligne (par caractère), du premier au dernier :

```
s = "Coucou tout le monde."
for c in s:
  print(c)
```

— Il existe donc deux manières principales pour parcourir les caractères d'une chaîne de caractères : soit (i) en itérant directement sur la chaîne elle-même (« `for c in s:` »), soit (ii) en itérant sur la séquence des indices dans la chaîne (« `for i in range(len(s)):` »). Par exemple, la suite d'instructions précédente est équivalente à celle-ci :

```
s = "Coucou tout le monde."
for i in range(len(s)):
  print(s[i])
```

— Lorsque l'on a besoin non seulement d'accéder aux caractères d'une chaîne mais aussi, simultanément, à leur position dans la chaîne, alors il est préférable d'itérer sur la séquence des indices

(« `for i in range(len(s)):` »). Lorsque l'on a seulement besoin d'accéder aux caractères, pas à leur position, alors il est peut-être plus simple d'itérer sur la chaîne elle-même (« `for c in s:` »).

Exercice 60 (Parcourir une chaîne de caractères, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
ch = "Avec consonnes"
cha = ch[0]
cha = cha + ch[2]
cha = cha + ch[6]
cha = cha + ch[9]
cha = cha + ch[12]
```

□

Exercice 61 (Modification d'une fonction, ⋆)
Considérer la suite d'instructions suivante :

```
# x: int
def message(x):
  return str(x)

a = 4
b = a * a
a = b - a
s = message(a)
print(s)
```

1. *Après exécution de cette suite d'instructions, que valent les différentes variables mentionnées (préciser leur type) ?*
2. *Modifier la fonction* `message` *de telle sorte que le programme affiche à la fin « Le résultat du calcul est* n*. » (où* n *est remplacé par la valeur de l'argument transmis à* `message`*).*
3. *Modifier la fonction* `message` *de telle sorte que le programme affiche à la fin «* $n \times n = n^2$ *» (où* n *et* n^2 *sont remplacés par la valeur de l'argument transmis à* `message` *et son carré respectivement).*

□

Exercice 62 (Itération par la fin, ⋆⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et affichant chaque caractère de* `s` *sur une ligne différente,* du dernier au premier.

Contrat :
Affichage pour `s="Salut"` *:*

```
t
u
l
a
S
```

□

Exercice 63 (Affichage des voyelles, ⋆)
On cherche à implémenter une fonction `print_vowels` *prenant en argument une chaîne de caractères* `s` *et affichant chaque voyelle minuscule de* `s` *dans l'ordre et sur une ligne chacune. Considérer la proposition suivante :*

```
# s: str
def print_vowels(s):
  for i in s:
    if(s[i] == "a" or "e" or "i" or "o" or "u" or "y"):
      print(s[i])
```

— *Expliquer pourquoi cette proposition ne convient pas (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `s` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

□

Exercice 64 (Compter les espaces, ⋆⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et retournant le nombre d'espaces contenus dans* `s`*. La fonction doit être construite autour d'une boucle for (sur les caractères de* `s`*).* □

Exercice 65 (Deux manières de parcourir une chaîne de caractères, ⋆⋆)
Écrire une suite d'instructions produisant exactement le même affichage que le code suivant mais itérant directement sur la chaîne `s` *définie plutôt que sur la séquence des indices* `range(len(s))`.

```
s = "sympa"
for i in range(len(s)):
  print(f"caractère en position {i} : {s[i]}")
```

□

Tranches [Cours]

— Une *tranche* d'une chaîne de caractères `s` est une sous-chaîne contiguë de `s`. Si, on accède à la tranche commençant à la position `i` (incluse) et se terminant à la position `j` (exclue) d'une chaîne `s` avec l'expression `s[i:j]`. Par exemple, si l'on a `s="Salut"`, `s[1:3]` vaut `"al"` — la tranche composée des caractères aux positions 1 et 2.
— `s[0:len(s)]` vaut toujours `s`.
— `s[i:i]` vaut toujours `""` (la chaîne vide).
— Si $0 \leq$ `i` $\leq$ `j` $\leq$ `len(s)`, `len(s[i:j])` vaut toujours `j-i`.
— Si $0 \leq$ `i` $<$ `len(s)`, alors `s[i:(i+1)]` vaut `s[i]`.
— Bien que Python autorise l'utilisation d'indices négatifs pour accéder à certaines tranches, cela est rarement une bonne idée, notamment parce que l'utilisation d'indices négatifs produit des résultats peu intuitifs. Par exemple, `"Salut"[-1:0]` vaut `""` et non `"t"` (c.-à-d. `"Salut"[-1]`). On a par contre que si $0 \leq$ `i` $<$ `j` $\leq$ `len(s)`, `s[(i-len(s)):j]` vaut `s[i:j]`.

Exercice 66 (Toutes les tranches, ⋆⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et affichant à l'écran toutes les tranches (non-vides) de* `s`*. La fonction doit être construite autour d'une double boucle for (définissant respectivement l'indice de début et l'indice de fin de la tranche à afficher).*

Contrat :
Affichage pour `s="Abba"` *:*

```
A
Ab
Abb
Abba
b
bb
bba
b
ba
a
```

□

Immutabilité des chaînes de caractères [Cours]

— Les chaînes de caractères sont *immutables*, c.-à-d. que l'on ne peut pas les modifier. Comme on l'a vu, il est possible de créer de nouvelles chaînes à partir de chaînes existantes, mais les chaînes existantes elles-mêmes ne sont jamais modifiées. Nous reviendrons sur la notion de (im)mutabilité dans un futur chapitre, mais l'on peut garder en tête que réaffecter une variable, c'est l'associer à un *nouvel* objet, et non modifier l'objet initialement associé à la variable. (Modifier une variable, ce n'est pas pareil que modifier une valeur.)

— Comme il n'est pas possible de modifier une chaîne de caractères, il est en particulier impossible de remplacer dans une chaîne un caractère à une position donnée par un autre. Par exemple, même si `s` désigne une chaînes de caractères non vide, « `s[0] = "c"` » n'est pas une instruction correcte ; l'exécution de cette instruction ne modifie pas le premier caractère de `s` mais lève l'exception « TypeError : 'str' object does not support item assignment ».

Exercice 67 (Remplacement du première caractère, ⋆⋆)
Écrire une fonction prenant en argument une chaîne de caractères `s` *et retournant* `None` *si* `s` *est de longueur nulle, et la chaîne obtenue en remplaçant son premier caractère par la version haut de casse de ce caractère sinon. Pour se faire, utiliser la notion de tranche.*

Contrat :
`s` = `"cinéma"` → *retour :* `"Cinéma"`

□

Exercice 68 (Remplacement d'un caractère quelconque, ⋆⋆)
Écrire une fonction prenant en argument une chaîne de caractères `s` *et un entier* `i`*, et retournant* `None` *si* `s` *est de longueur strictement inférieure à* `i`*, et la chaîne obtenue en remplaçant le caractère de position* `n` *par sa version haut de casse sinon.*

Contrat :
`s, i` = `"cinéma", 2` → *retour :* `"ciNéma"`

□

Afficher du texte dans le terminal : `print` [Cours]

— La fonction `print` peut prendre un nombre quelconque d'arguments (c.-à-d. zéro, un ou plus).
— Pour appeler une fonction sans arguments, il faut quand même écrire les parenthèses.
— La fonction `print` parcourt tous ses arguments de gauche à droite (hors arguments spéciaux `end` et `sep` qui sont évoqués ci-dessous) et les affiche l'un après l'autre dans le terminal. Par défaut, l'affichage des arguments est séparé par un espace (`" "`) et se termine par un saut à la ligne (`"\n"`).
— Par exemple, la suite d'instructions suivante :

```
print("Hello", "world!") # Affiche "Hello world!" et passe à la ligne.
print() # Passe à la ligne.
```

produit l'affichage suivant et passe à la ligne (ce qui signifie que tout nouvel affichage commencera sur une troisième ligne) :

```
1. Hello world!
2.
```

— La chaîne qui termine l'affichage de `print` — par défaut un saut de ligne, `"\n"` — est contrôlable via un argument spécial : `end`. Si l'on veut terminer l'affichage par une chaîne `s` (plutôt que par `"\n"`), il faut appeler `print` en indiquant `end=s` dans la liste d'arguments, après les arguments « normaux » à afficher.
— Par exemple, la suite d'instructions suivante :

```
print("Hello world!", end="") # Pas de saut de ligne.
print("Hello world!", end="\n\n") # Deux sauts de ligne.
print("Hello world!") # Par défaut, un saut de ligne.
```

produit l'affichage suivant et passe à la ligne (ce qui signifie que tout nouvel affichage commencera sur une quatrième ligne) :

```
1. Hello world!Hello world!
2.
3. Hello world!
```

— En exercice, lorsque l'on vous demande quel est l'affichage produit par l'exécution d'un programme, indiquez clairement toutes les lignes, même les lignes vides.
— Notons que n'importe quelle chaîne de caractères peut être passée comme argument `end` à la fonction `print`, pas seulement une chaîne composée de retours à la ligne.
— On peut également modifier ce qui est affiché entre les arguments à l'aide d'un autre argument spécial : `sep`. Si l'on veut que l'affichage de chaque argument soit séparé par une chaîne `s` (plutôt que par `" "`), il faut appeler `print` en indiquant `sep=s` dans la liste d'arguments, après les arguments « normaux » à afficher.
— Par exemple, la suite d'instructions suivante :

```
print("Hello", "world!", sep="-")
print("Hello", "world!", sep="  ")
```

produit l'affichage suivant et passe à la ligne (ce qui signifie que tout nouvel affichage commencera sur une troisième ligne) :

```
1. Hello-world!
2. Hello  world!
```

Exercice 69 (Comprendre les arguments de print, ⋆⋆)
Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?

```
print("a", "b")
print("c", "d", end="")
print("e", "f", sep="|")
print("g", "h", end="|")
print("i", "j", end="|\n")
print("k", "l", sep="\n")
print(end="")
```

□

Exercice 70 (Expliciter les arguments par défaut de print, ⋆)
Donner une instruction équivalente à `print("Hello", "World!")` *dans laquelle les arguments* `sep` *et* `end` *de* `print` *sont donnés explicitement.* □

Exercice 71 (Réécritures, ⋆⋆)

1. *Écrire une fonction qui prend en argument une chaîne de caractères* `s` *et affiche une suite de tirets (-) de même longueur. La fonction ne doit pas faire appel à la méthode* `replace`*, mais doit être construite autour d'une boucle faisant des appels à* `print`*.*

   Contrat :
   `s = "cinéma"` → *affichage :* `------`

2. *Écrire une fonction qui prend en argument une chaîne de caractères et l'affiche à l'envers.*

   Contrat :
   `s = "cinéma"` → *affichage : aménic*

□

Exercice 72 (Affichage sans espaces, ⋆)

1. *Écrire une fonction qui prend en argument une chaîne de caractères* `s` *et affiche cette même chaîne sans les espaces. La fonction ne doit pas faire appel à la méthode* `replace`*.*

   Contrat :
   `s = "Bonjour tout le monde !"` → *affichage : Bonjourtoutlemonde !*

2. *Même question, mais en utilisant la méthode* `replace`*.*

□

Récupérer du texte depuis le terminal : `input` [Cours]

— Il existe une fonction qui est en quelque sorte l'inverse de `print` : `input`. Cette fonction permet de récupérer une saisie au clavier.
— L'évaluation de l'expression `input()` demande une saisie au clavier. La saisie s'arrête avec l'appuie sur la touche d'entrée et la valeur de l'expression `input()` est une chaîne de caractères, la séquence correspondant aux touches tappées jusqu'à la touche d'entrée (non incluse).
— La chaîne retournée ne contient jamais de retour à la ligne (`"\n"`) et c'est pourquoi on parle d'une *ligne de texte*. (Il est tout à fait possible de saisir les caractères `"\"` suivi de `"n"`, la chaîne retournée contiendra cette séquence de deux caractères — noté `"\\n"`, en échappant la barre oblique inverse — et non le caractère `"\n"`.)
— Généralement (c.-à-d. sauf pour certaines utilisations d'`input` en débuggage de code), on souhaite faire quelque chose avec le texte saisi au clavier. Il est donc très courant de stocker la chaîne retournée par `input`, comme dans le code suivant :

```
s = input()

print(f"Le texte entrée est de longueur {len(s)}.")
#print("Le texte entrée est de longueur " + str(len(s)) + ".")

if(len(s) > 0): print(f"Le premier caractère saisi est '{s
    [0]}'.").
#if(len(s) > 0): print("Le premier caractère saisi est '" + s
    [0] + "'.").
else: print("La chaîne est vide, il n'y a donc pas eu de
    premier caractère saisi.")
```

— La valeur de retour de la fonction `input` est toujours de type `str` (c.-à-d. qu'il s'agit d'une chaîne de caractères). Cependant, il est possible d'utiliser ensuite la fonction `int` ou la fonction `float` pour convertir cette valeur en entier ou nombre à virgule :

```
n = int(input())
print(f"2 x n = {2 * n}")
```

— Un message d'invitation à la saisie peut être passé en argument, comme dans `input("Veuillez entrer une phrase ci-dessous.\n")`. Autrement dit, les deux blocs de code suivants sont équivalents :

```
print("Comment vous appelez-vous ?\n")
name = input()
```

```
name = input("Comment vous appelez-vous ?\n")
```

3.4 Exercices supplémentaires

Exercice 73 (Concaténation, ⋆)
Écrire une fonction qui prend en argument une chaîne `s` *de caractères et affiche une ligne commençant par « Bonjour, » suivi de* `s`.

Contrat :
`s = "vous allez bien ?"` → *affichage :* `Bonjour, vous allez bien ?` *(puis saut de ligne)*

□

Exercice 74 (Concaténation et somme, ⋆⋆)
Après exécution de la suite d'instructions suivante, que valent les différentes variables mentionnées ?

```
x = 0
s = ""
for n in range(10):
    x = x + 1
    s = s + "1"
```

□

Exercice 75 (Ordinaux anglais, ⋆⋆⋆)
On s'intéresse aux ordinaux anglais abrégés, *où le nombre (le cardinal) est écrit en chiffres suivi d'un suffixe :*

1st, 2nd, 3rd, 4th, ..., 9th, 10th, 11th, 12th, ..., 19th, 20th, 21st, 22nd, 23rd, ...

Règle (sauf exception) : Si le dernier chiffre du nombre est 1, le suffixe est « st » ; si c'est 2, le suffixe est « nd » ; si c'est 3, le suffixe est « rd » ; sinon le suffixe est « th ».
Exception : Si l'avant-dernier chiffre du nombre est 1, le suffixe est toujours « th ».

Écrire une fonction `get_ordinal` *qui prend un entier de type* `int` *en argument et retourne l'ordinal anglais abrégé correspondant (une* `str`*). Il peut s'avérer utile d'utiliser la fonction* `str` *pour obtenir la représentation textuelle d'un entier.*

Contrat :
1 → *retour : 1st*
12 → *retour : 12th*
23 → *retour : 23rd*
32 → *retour : 32nd*
44 → *retour : 44th*

□

Exercice 76 (Réécriture, ⋆⋆)
Écrire une fonction qui prend en argument une chaîne de caractères `s` *et affiche une suite de tirets (-) de même longueur. La fonction doit faire appel de manière pertinente à la méthode* `replace`.

Contrat :
`s = "cinéma"` → *affichage :* `------`

□

Exercice 77 (Lignes et pointillés, ⋆⋆)

1. *Écrire une fonction qui prend en argument un entier supposé positif* `n` *et qui affiche une ligne d'astérisques « * » de longueur* `n` *puis va à la ligne.*

   Contrat :
   `n = 7` → *affichage :* `*******` *(plus saut de ligne)*

2. *Modifier la fonction pour qu'elle affiche une ligne de longueur* `n` *commençant par un astérisque et alternant astérisques et tirets bas « _ ».*

Contrat :
`n = 7` → *affichage :* `*_*_*_*` *(plus saut de ligne)*

3. *Qu'affiche la dernière version de la fonction si* `n` *est pair ?*

□

Exercice 78 (Toutes les tranches, ⋆⋆⋆)

1. *Considérant une chaîne* `s` *de caractères et* `i` *une position dans cette chaîne (avec* $0 \leq$ `i` $<$ `len(s)`*). Quelle est, en fonction de* `len(s)` *et de* `i`*, la longueur de la plus longue tranche de* `s` *commençant à l'indice* `i` *?*
2. *Écrire une fonction prenant en argument une chaîne* `s` *de caractères et affichant à l'écran toutes les tranches (non-vides) de* `s`*. La fonction doit être construite autour d'une double boucle for : l'une sur les indices dans* `s` *et l'autre sur les longueurs des tranches commençant à ces indices.*

□

Exercice 79 (Occurrences d'un caractère, ⋆⋆)
Écrire une fonction `print_where` *qui prend en argument une chaîne de caractères* `s` *et un caractère* `c` *(c.-à-d. une chaîne de caractères supposée de longueur 1) et affiche toutes les positions de* `s` *auquelles* `c` *apparaît.*

Contrat :
(s,c) = ("abracadabra","a") → *affichage : 0 3 5 7 10*

□

Exercice 80 (Pyramide, ⋆⋆⋆)
Écrire une fonction prenant en argument un entier `n`*, et affichant une pyramide d'étoiles de hauteur* `n` *si ce nombre est positif et une pyramide d'étoiles inversée de hauteur* `-n` *sinon.*

Contrat :
Affichage pour $n = 4$ *:*

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

Affichage pour $n = -3$ *:*

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

□

Chapitre 4

Retour sur les boucles

Objectifs :

- Manipuler des boucles avec accumulateurs.
- Manipuler des boucles imbriquées.
- Définir des boucles `while`.
- Contrôler finement l'exécution des différents types de boucles avec les instructions `break`, `continue` et `return`.

4.1 Utilisation avancée des boucles

Boucles *for* et accumulateurs [Cours]

— Au premier chapitre, nous avons vu comment modifier la valeur d'une variable avec une (ré)affectation. Par exemple, après exécution du code suivant, la valeur de la variable `n` est 43 :

```
n = 42
n = n + 1
```

— Il est tout à fait possible d'incrémenter une variable dans le corps d'une boucle, ce qui répète la modification. Dans ce cas, il ne faut pas oublier d'initialiser la variable avant la boucle.

```
n = 42
for i in range(10):
  n = n + 1
```

— Quand la valeur d'une variable est construite en « ajoutant un morceau » à chaque itération d'une boucle, comme dans l'exemple précédent, on parle d'un *accumulateur*. Cet ajout d'un morceau peut se faire de façon différente selon le problème à résoudre, par exemple par des opérations arithmétiques, par la concaténation de chaînes de caractères, ou encore d'autres opérations (voir l'exercice 83).

— Par exemple, le programme suivant calcule dans `s` la somme des carrés des entiers de 1 à 100 (inclus) :

```
s = 0
for i in range(1, 101):
  s = s + (i * i)
```

— Au passage, pour beaucoup d'opérateurs `@` (par exemple +, −, ∗), l'assignation `x = x @ y` peut aussi s'écrire `x @= y`. Par exemple, le programme suivant est équivalent au précédent :

```
s = 0
for i in range(1, 101):
  s += (i * i)
```

Exercice 81 (Accumulation, ⋆)
Modifier la boucle donnée ci-dessus pour qu'elle calcule la somme des cubes des entiers de 1 à 25 (inclus). □

Exercice 82 (Comprendre et modifier une boucle, ⋆)
Considérer la suite d'instructions suivante :

```
st = ""
for i in range(50):
  st = st + "ab"
```

1. *Que contient la variable* `st` *après exécution de cette suite d'instructions ?*
2. *Modifier les instructions ci-dessus pour qu'à la fin de leur exécution la variable* `st` *contienne la chaîne de caractère* `"aaaaaa ... aaaa"` *avec la lettre « a » répétée 110 fois.*
3. *Modifier de nouveau la suite d'instructions pour imprimer à l'écran le contenu de la variable* `st` *à chaque tour de boucle. Quel est alors l'affichage produit par l'exécution de la suite d'instructions ?*

□

Exercice 83 (Accumulation booléenne, ⋆⋆)
Rappelons que la fonction `input` *permet de récupérer une ligne de texte saisie au clavier. Si l'on veut lire un entier tapé au clavier, il suffit de convertir la chaîne récupérée avec la fonction* `int`*.*

Écrire une fonction prenant en argument un entier `n` *et qui (i) lit* `n` *entiers au clavier, puis (ii) affiche « Gagné » si l'entier 42 se trouve parmi ceux-là, et « Perdu » sinon. (Attention, il faut d'abord lire les* `n` *entiers et seulement ensuite afficher le résultat.)* □

Boucles imbriquées [Cours]

— Le corps d'une boucle peut lui-même contenir une boucle. On parle alors de *boucles imbriquées*.
— Par exemple, le code suivant

```
for i in range(3):
  for j in range(5):
    print(f"i = {i}, j = {j}")
```

affiche :

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 0, j = 3
i = 0, j = 4
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 1, j = 4
i = 2, j = 0
i = 2, j = 1
i = 2, j = 2
i = 2, j = 3
i = 2, j = 4
```

— Un nombre arbitrairement grand de boucles peuvent être imbriquées les unes dans les autres.
— La boucle la plus interne/profonde est « la plus rapide », et la boucle la plus externe/moins profonde est « la moins rapide » ;
— Chaque boucle a son propre itérable et son propre compteur.

Exercice 84 (Cent, ⋆⋆)
Le but de cet exercice est d'écrire un programme affichant les entiers de 0 à 99 en 10 lignes de 10 nombres séparés par des espaces :

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
...
90 91 92 93 94 95 96 97 98 99
```

1. *Écrire une fonction* `loop_aux` *prenant en argument un entier* `i` *et affichant la* `i+1`*-ième ligne de l'affichage décrit en introduction de cet exercice. La fonction doit être construite autour d'une boucle for.*

   Contrat :
   `i = 2` → *affichage :* `20 21 22 23 24 25 26 27 28 29`

2. *Écrire un programme utilisant* `loop_aux` *pour produire la totalité de l'affichage décrit en introduction de cet exercice.*
3. *Réécrire le programme précédent sans faire appel à* `loop_aux` *et de manière à ce qu'il soit construit autour d'une double boucle. Pour chacune des boucles, indiquer en commentaire si une de ses itérations correspond à l'affichage d'une ligne complète ou bien d'un fragment de ligne.*
4. *Réécrire complètement le programme précédent de manière à ce qu'il soit construit autour d'une simple boucle (incluant une structure conditionnelle pour gérer les sauts de ligne).*

□

Exercice 85 (Triangle, ⋆⋆)
Le but de cet exercice est d'écrire une fonction prenant en argument un entier `n` *et affichant* `n` *lignes telles que la* `i`*-ième ligne est constituée de* `i` *étoiles (« * ») séparées par des espaces. Par exemple, pour* `n=5` *:*

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

1. *Écrire une fonction* `loop_aux` *prenant en argument un entier* `i` *et affichant une ligne composée de* `i+1` *étoiles séparées par des espaces. La fonction doit être construite autour d'une boucle for.*
2. *Écrire une fonction* `triangle` *prenant en argument un entier* `n` *et utilisant* `loop_aux` *pour produire l'affichage décrit en introduction de cet exercice.*
3. *Réécrire* `triangle` *sans faire appel à* `loop_aux` *mais de manière à ce que la fonction soit construite autour d'une double boucle. Pour chacune des boucles, indiquer en commentaire si une de ses itérations correspond à l'affichage d'une ligne complète ou bien d'un fragment de ligne.*
4. *Réécrire* `triangle` *de manière à ce que la fonction soit construite autour d'une simple boucle (et en pensant soit à la répétition* `*`*, soit à la concaténation* `+` *de chaîne).*

□

Boucles *while* [Cours]

— En Python, une boucle *while* (aussi appelée « boucle non bornée ») se définit par :
  — une expression booléenne — la *condition* de la boucle ;
  — un bloc de code — le *corps* de la boucle.
— Une boucle *while* s'écrit à l'aide du mot-clef `while` suivi de la condition notée entre parenthèses et suivi d'un deux-points. Le corps est indiqué sur les lignes suivantes, indenté par rapport au mot-clef `while`.

— La boucle suivante affiche les entiers pairs de 0 (inclus) à 10 (exclu) :

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

— Les boucles *while* permettent d'exécuter en boucle un bloc de code tant qu'une certaine condition est satisfaite. Plus précisément, l'exécution d'une boucle *while* se déroule de la manière suivante :

1. la condition est évaluée ; si elle vaut `False`, alors l'exécution de la boucle est terminée, sinon, on passe à l'étape 2 ;
2. le corps de la boucle est exécuté avant de retourner à l'étape 1.

— Les boucles *while* sont notamment utiles lorsque l'on doit exécuter en boucle un bloc de code sans connaître *a priori* le nombre d'itérations nécessaires. Par exemple, il n'est pas évident de savoir comment obtenir le même affichage que le code suivant en utilisant une boucle *for* plutôt qu'une boucle *while* :

```
i = 0
while((((i * 17) - 11 // 3) % 7) != 6):
  print(f"Pour i={i}, la condition est encore vérifiée.")
  i = i + 1
print(f"Pour i={i}, la condition n'est pas vérifiée.")
```

— Il se trouve que dans l'exemple précédent, le corps de la boucle est exécuté quand `i` prend successivement les valeurs 0, 1 et 2 (quand `i` prend la valeur 3, la condition n'est plus vérifiée et la boucle se termine). On pourrait donc réécrire la boucle précédente en une boucle *for* sur la séquence `range(3)`. Il est cependant parfois impossible de transformer une boucle *while* en une boucle *for*, tout simplement parce que l'itération ne se fait pas sur une séquence de valeurs pouvant être connue a priori. C'est par exemple le cas lorsque l'on souhaite exécuter un bloc de code tant que tant que l'utilisateur·rice n'a pas effectué une certaine action donnée (définie typiquement avec la fonction `input`) :

```
s = input("Bonjour.")
is_polite = s.lower().startswith("bonjour")
while(not is_polite):
  s = input("...")
  is_polite = s.lower().startswith("bonjour")
print("Comment allez-vous ?")
```

— L'exécution d'une boucle *while* peut ne jamais terminer. C'est potentiellement le cas dans l'exemple précédent, mais aussi toujours le cas dans l'exemple suivant :

```
i = 0
while(True):
  print(i)
  i += 1
print("Ceci ne s'affichera jamais.")
```

Exercice 86 (Première boucle while, ⋆)
Après exécution de la suite d'instructions suivante, que vaut `r` *?*

```
n = 25
r = 0
while((r * r) < n):
  r = r + 1
```

□

Exercice 87 (Les boucles for vues comme des boucles while, ⋆)
Réécrire la suite d'instructions suivante pour obtenir le même comportement en utilisant une boucle while à la place de la boucle for.

```
a = 0
for i in range(32):
  a = a + i
  print(a)
```

□

Exercice 88 (Terminaison, ⋆)
Qu'affichent les suites d'instructions suivantes ? Est-ce que leur exécution se termine ?

1.
```
n = 2
while(n > 0):
  print(n)
```

2.
```
n = 2
while(n > 0):
  n = n * 2
  print(n)
```

3.
```
n = -5
while(n != 0):
  n = n + 1
  print(n)
```

4.
```
n = -5
while(n != 0):
  n = n + 2
  print(n)
```

□

Exercice 89 (Logarithme (en base 2), ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et renvoyant le nombre de fois qu'il faut diviser celui-ci par deux avant que le résultat ne soit inférieur ou égal à 1. La fonction doit être construite autour d'une boucle while.*

Contrat :
`n` = 1 → *retour :* 0
`n` = 2 → *retour :* 1
`n` = 3 → *retour :* 2
`n` = 4 → *retour :* 2
`n` = 5 → *retour :* 3

□

Exercice 90 (L'ordinateur obstiné, ⋆)
Écrire un programme qui demande à l'utilisateur·rice de saisir un entier au clavier (en utilisant `input`*) et demande une nouvelle valeur tant que la précédante n'est pas paire. Quand un entier saisi est pair, le programme doit afficher « Merci. » et terminer.* □

Contrôle fin des boucles [Cours]

— Il est possible de « sortir » d'une boucle (*for* comme *while*), c.-à-d. d'arrêter son exécution, à n'importe quel moment à l'aide de l'instruction `break`.

— Par exemple, les trois codes suivants sont équivalents :

```
for _ in range(100):
  s = input("Saisissez le nom de la capitale du Canada.\n")
  if(s.lower() == "ottawa"):
    print("Merci.")
    break

print("Au revoir.")
```

```
i = 0
while(True):
  s = input("Saisissez le nom de la capitale du Canada.\n")
  if(s.lower() == "ottawa"):
    print("Merci.")
    break

  i = i + 1
  if(i >= 100):
    break

print("Au revoir.")
```

```
i = 0
stop = False
while(not stop):
  s = input("Saisissez le nom de la capitale du Canada.\n")
  if(s.lower() == "ottawa"): print("Merci.")

  i = i + 1
  stop = (i >= 100) or (s.lower() == "ottawa")

print("Au revoir.")
```

— Il est possible de terminer un tour de boucle sans sortir de la boucle, c.-à-d. de passer directement au tour suivant, à l'aide de l'instruction `continue`.

— Par exemple, les deux codes suivants sont équivalents :

```
for i in range(100):
  print(i)
  if((i % 3) == 0):
    print(f"{i} est un multiple de 3.")
    if((i % 4) == 0):
      print("C'est aussi un multiple de 4.")
```

```
for i in range(100):
  print(i)

  if(not ((i % 3) == 0)): continue
  print(f"{i} est un multiple de 3.")

  if(not ((i % 4) == 0)): continue
  print("C'est aussi un multiple de 4.")
```

— Il est important de garder en tête que l'exécution d'une fonction s'arrête dès qu'une instruction de retour est rencontrée. Ainsi, si une boucle apparaît dans le corps d'une fonction, une instruction de retour rencontrée dans le corps de la boucle termine l'exécution de la fonction et donc aussi celle de la boucle, ce qui rappelle l'effet d'une instruction `break`. Par exemple, l'exécution du code suivant affiche seulement « avant la boucle », « début de l'itération i=0 », « fin de l'itération i=0 », « début de l'itération i=1 », « fin de l'itération i=1 » et « début de l'itération i=2 » :

```
# n: int
def f(n):
  print("avant la boucle")
  for i in range(n):
    print(f"début de l'itération i={i}")
    if(i == 2): return i
    print(f"fin de l'itération i={i}")
  print("après la boucle")

f(12)
```

Exercice 91 (if contre continue, ⋆⋆)

1. *Écrire une fonction qui prend en argument deux entiers* `n` *et* `k`*, et qui parcourt les entiers de 2 (inclus) à* `n` *(exclu) en affichant tous ceux qui divisent* `k`*. (Rappelons que* `((i % j) == 0)` *teste si* `i` *est divisé par* `j` *; autrement dit, si* `j` *divise* `i`*.) La fonction doit être construite autour d'une boucle contenant une structure conditionnelle et ne doit pas utiliser* `continue`*.*

   Contrat :
   Affichage pour `n=8` *et* `k=24` *:*

   2
   3
   4
   6

2. *Même question mais en utilisant* `continue`*.*

□

Exercice 92 (break contre return, ⋆⋆)

1. *Écrire une fonction qui prend en argument deux entiers* `n` *et* `k`*, et qui parcourt les entiers de 2 (inclus) à* `n` *(exclu) en affichant uniquement le premier qui divise* `k` *(s'il existe). La fonction doit utiliser une instruction* `break`*.*

   Contrat :
   `n, k = 9, 24` → *affichage :* 2
   `n, k = 9, 35` → *affichage :* 5
   `n, k = 5, 35` → *(aucun affichage)*

2. *Même question mais le premier diviseur de* `k` *trouvé doit non pas être affiché mais renvoyé et sans utiliser d'instruction* `break`*. (Si aucun diviseur de* `k` *n'est trouvé, la fonction doit retourner* `None`*.)*

□

Exercice 93 (Uniquement des zéros et des uns, ⋆⋆)
On cherche à implémenter une fonction `binary_string` *prenant en argument une chaîne de caractères* `s` *et renvoyant* `True` *si* `s` *n'est composée que des caractères* `"0"` *et* `"1"`*, et* `False` *sinon. Considérer la proposition suivante.*

```
# s: str
def binary_string(s):
  for c in s:
    if((c == 0) or (c == 1)):
      return True
    else:
      return False
```

— *Expliquer pourquoi cette proposition est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `s` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

□

Exercice 94 (Palindrome, ⋆⋆)
Un mot est un palindrome s'il peut s'écrire indifféremment de la gauche vers la droite ou de la droite vers la gauche. Écrire une fonction prenant en argument une chaîne de caractères `s` *et renvoyant* `True` *si* `s` *est un palindrome et* `False` *sinon.*

Contrat :
`s = "kayak"` → *retour :* `True`
`s = "abba"` → *retour :* `True`
`s = "abc"` → *retour :* `False`

□

Exercice 95 (Jeu, ⋆⋆⋆)
Écrire une fonction `guess_number` *qui prend en argument un entier* n *supposé positif, sélectionne un entier au hasard entre 0 et* n *inclus avec l'instruction* `target = random.randint(0, n)`*, puis laisse un nombre illimité de tours à l'utilisateur·rice pour deviner la valeur de* `target`*. À chaque tour, la fonction* `input` *est utilisée pour que l'utilisateur·rice saisisse un entier ; ensuite, doit être indiqué si la valeur entrée est la bonne ou si elle est plus haute ou plus basse que* `target`*. La fonction s'arrête dès que la bonne valeur est entrée et après avoir affiché le nombre de tours effectués. (En pratique, pour pouvoir utiliser la fonction* `random.randint`*, il faut avoir au préalable exécuté l'instruction* `import random`*. Il est par exemple possible de mettre cette instruction avant la définition de la fonction.)* □

4.2 Exercices supplémentaires

Exercice 96 (Puissance, ⋆)
Écrire une fonction `power` *construite autour d'une simple boucle et telle que* `power(x, n)` *renvoie la valeur* x^n *(c.-à-d.* $x \times x \times \cdots \times x$*).* □

Exercice 97 (Factorielle, ⋆)
Écrire une fonction `fact` *construite autour d'une simple boucle et telle que* `fact(n)` *renvoie la factorielle de* n *(c.-à-d.* $1 \times 2 \times \cdots \times n$*).* □

Exercice 98 (Double triangle, ⋆⋆)
Écrire une fonction `double_triangle` *qui a comme argument un entier* `n` *et qui affiche* `n+1` *lignes telles que la* `i`*-ième ligne est constituée de* `(i-1)` *« * » et* `(n+1-i)` *« 0 », tous séparés par des espaces.*

Contrat :
Affichage pour `n=5` *:*

```
0 0 0 0 0
* 0 0 0 0
* * 0 0 0
* * * 0 0
* * * * 0
* * * * *
```

□

Exercice 99 (Ça use, ⋆⋆)

1. *Écrire un programme qui affiche les paroles de la fameuse chanson (de 1 à 100 kilomètres ; attention au singulier au début de la chanson) :*

```
1 kilomètre à pied, ça use, ça use,
1 kilomètre à pied, ça use les souliers.
2 kilomètres à pied, ça use, ça use,
2 kilomètres à pied, ça use les souliers.
...
100 kilomètres à pied, ça use, ça use,
100 kilomètres à pied, ça use les souliers.
```

2. *Modifier le code pour qu'il affiche les (paires de) vers dans l'ordre inverse.*

□

Exercice 100 (Nombres premiers, ⋆⋆)

1. *Écrire une fonction* `is_not_prime` *prenant en argument un entier* `n` *et renvoyant* `True` *si* `n` *n'est pas premier (c.-à-d. s'il est inférieur ou égal à 1 ou s'il existe un entier strictement entre 1 et* `n` *divisant* `n`*) et* `False` *sinon. La fonction doit être construite autour d'une boucle for.*

Contrat :
`n = 1` → *retour :* `True`
`n = 2` → *retour :* `False`
`n = 3` → *retour :* `False`
`n = 4` → *retour :* `True`
`n = 12` → *retour :* `True`
`n = 17` → *retour :* `False`

2. *Écrire une fonction* `sum_primes` *telle que* `sum_primes(n)` *renvoie la somme des nombres premiers compris entre* `1` *et* `n` *(inclus). La fonction doit être construite autour d'une boucle for.*

Contrat :
`n = 10` → *retour :* `4`
`n = 20` → *retour :* `8`

3. *Écrire une fonction* `next_prime` *prenant en argument un entier* `n` *et renvoyant le plus petit nombre premier supérieur ou égal à* `n`. *La fonction doit être construite autour d'une boucle while.*

   Contrat :
   `n = 2` → *retour :* `2`
   `n = 10` → *retour :* `11`
   `n = 20` → *retour :* `23`

□

Exercice 101 (Table de multiplication, ⋆⋆)

1. *Écrire un programme qui affiche les tables de multiplication pour les entiers de* 1 *à* 10.

```
1 2 3 4 ... 10
2 4 6 8 ... 20
...
10 20 30 ... 100
```

2. *Modifier le programme précédent de manière à écrire chaque valeur dans une case de trois caractères de large (la taille nécessaire pour écrire « 100 »), plus un espace supplémentaire entre les cases.*

```
1   2   3   4   ... 10
2   4   6   8   ... 20
...
10  20  30  40  ... 100
```

3. *Écrire une fonction* `in_table` *telle que* `in_table(n)` *renvoie* `True` *si* n *apparaît dans l'une des tables de multiplication de 1 à 10 et* `False` *sinon.*

□

Exercice 102 (Carrés, ⋆⋆)

1. *Écrire une fonction qui a comme argument un entier supposé positif* n *et qui affiche un carré d'étoiles plein de côté* n.

   Contrat :
   Affichage pour $n = 4$ *:*

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

2. *Modifier la fonction pour qu'elle affiche un carré creux.*

   Contrat :
   Affichage pour $n = 4$ *:*

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

□

Exercice 103 (Sous-chaîne, ⋆⋆)

1. *Écrire une fonction* `contain` *qui prend en argument deux chaînes de caractères* `s` *et* `ss` *(cette dernière est supposée non vide) et retourne* `True` *si* `ss` *apparaît comme sous-chaîne dans* `s` *et* `False` *sinon. Votre code doit être construit autour d'une double boucle, l'une itérant sur les indices dans* `s` *à partir desquels on se demande si* `ss` *apparaît, et l'autre sur les indices dans* `ss`.

   Contrat :
   (s, ss) = ("abracadabra", "cada") → *retour :* `True`
   (s, ss) = ("abracadabra", "cadb") → *retour :* `False`

2. *Écrire une fonction* `contain_where` *qui prend en argument deux chaînes de caractères* `s` *et* `ss` *(cette dernière est supposée non vide), et affiche toutes les positions de* `s` *à partir desquelles* `ss` *apparaît comme sous-chaîne. Votre code doit être construit autour d'une double boucle.*

Contrat :
(s, ss) = ("abracadabra", "abra") $\rightarrow$ *affichage : 0 7*

□

Exercice 104 (Racine cubique, ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et renvoyant le plus petit entier* a *tel que* $a^3 \geq n$.

Contrat :
`n = 1` $\rightarrow$ *retour :* `1`
`n = 8` $\rightarrow$ *retour :* `2`
`n = 27` $\rightarrow$ *retour :* `3`
`n = 2` $\rightarrow$ *retour :* `2`
`n = 9` $\rightarrow$ *retour :* `3`
`n = 28` $\rightarrow$ *retour :* `4`

□

Exercice 105 (Syracuse, ⋆⋆⋆)
La suite de Syracuse de premier terme p *(entier strictement positif), notée* $(u_{p,n})_{n\in\mathbb{N}}$, *est définie par récurrence par* $u_{p,0} = p$ *et*

$$\forall n \in \mathbb{N},\ u_{p,n+1} = \begin{cases} \frac{u_{p,n}}{2} & \text{si } u_{p,n} \text{ est pair} \\ 3u_{p,n} + 1 & \text{si } u_{p,n} \text{ est impair} \end{cases}$$

Une conjecture (encore non prouvée à ce jour) est que pour tout entier strictement positif p, *la suite de Syracuse de premier terme* p *finit par atteindre 1, c.-à-d. qu'il existe* $n \in \mathbb{N}$ *tel que* $a_{p,n} = 1$. *Écrire une fonction prenant en argument un entier* `p` *et retournant le premier entier* n *telle que* $u_{p,n} = 1$. □

Chapitre 5

Immutabilité, listes et n-uplets

Objectifs :

- Comprendre le concept de mutabilité et d'immutabilité.
- Définir et décomposer des *n*-uplets.
- Définir des listes en extension et en intension/-compréhension.
- Savoir modifier des listes, en réassignant directement l'élément à un indice donné et avec des méthodes comme `append`, `extend` ou `pop`.
- Comprendre la différence entre opérations fonctionnelles et non fonctionnelles.
- Savoir utiliser les fonctions `enumerate`, `zip`, `sorted`, `sort`, `reversed` et `reverse`.

5.1 Immutabilité et *n*-uplets

Immutabilité [Cours]

- Une valeur *immutable* est une valeur que l'on ne peut pas altérer.
- Tous les types de valeurs vus jusqu'à présent (`int`, `str`, etc.) sont des types de valeurs immutables.
- Prenons l'exemple des chaînes de caractères.
  - Il est possible de réassigner une variable contenant une chaîne de caractères, mais cela ne modifie pas la chaîne elle-même (seulement la variable). Par exemple :

```
s1 = "Bonjour."
s2 = s1
s2 = "Au revoir."
print(s2) # Affiche "Au revoir.".
print(s1) # Affiche "Bonjour.".
```

  - Il est possible de créer une chaîne de caractères à partir d'une autre avec, par exemple, la méthode `lower`, mais cela ne modifie pas la chaîne originale. Par exemple :

```
s1 = "Bonjour."
s2 = s1.lower()
print(s2) # Affiche "bonjour.".
print(s1) # Affiche "Bonjour.".
```

- Rappelons-nous que durant l'exécution d'un programme Python, la machine maintient en mémoire les associations entre noms de variables et valeurs, et que l'on peut représenter visuellement ces associations en dessinant d'un côté un ensemble de noms de variables, de l'autre un ensemble de valeurs, et, pour chaque nom de variables, une flèche partant du nom de variable et pointant sur l'une des valeurs.
  - Si `s1` est un nom de variable non encore définie, alors l'exécution de `s1 = "Bonjour."` ajoute un `"Bonjour."` du côté des valeurs, et `s1` du côté des variables avec une flèche de ce `s1` vers

cette valeur.
  - Ensuite, si `s2` est un nom de variable elle aussi non encore définie, l'exécution de `s2 = s1.lower()` ajoute un `"bonjour."` (la valeur de `s1.lower()`) du côté des valeurs, et `s2` du côté des variables avec une flèche de ce `s2` vers cette valeur.
- Comme nous le verrons dans la suite du cours, il existe en PYTHON des valeurs *mutables*, c.-à-d. que l'on peut modifier. Modifier une valeur mutable, ce n'est pas changer les associations entre variables et valeurs (les flèches), mais modifier directement une valeur.
- Imaginons qu'une fonction `mutable_str` permette d'accéder à un type de chaînes de caractères mutables. On pourrait imaginer que ce type possède une méthode `mutable_lower` qui modifie directement la valeur sur laquelle est appelée, en la passant en bas de casse. Voici ce que l'on pourrait observer :

```
s1 = mutable_str("Bonjour.")
s2 = s1
s1.mutable_lower()
print(s2) # Affiche "bonjour.".
```

Exercice 106 (Valeurs et variables, ⋆)
Simuler à la main l'exécution du code suivant instruction par instruction tout en maintenant à jour les associations entre noms de variable et valeurs.

```
x = 1
y = x
x += 1
x += 1
```

□

Le type `tuple` [COURS]

- Le type `tuple` est le type des structures de données appelées « *n*-uplets » en PYTHON. Un *n*-uplet est une suite finie et ordonnée de valeurs appelées « éléments ». Les *n*-uplets constitués de deux éléments sont appelés « paires ».
- Le *n*-uplet vide s'écrit « `()` » ou, de manière équivalente, `tuple()`.
- Le *n*-uplet contenant uniquement la valeur de `x1` s'écrit « `(x1,)` ». La virgule est très importante, car `(x1)` n'est pas un *n*-uplet mais vaut tout simplement `x1`. En revanche, `x1` et `(x1,)` sont deux objets différents. Par exemple :

```
print(3) # Affiche "3".
print((3)) # Affiche "3".
print(3 == (3)) # Affiche "True".
print((3,)) # Affiche "(3,)".
print(3 == (3,)) # Affiche "False".
```

- La paire constituée des valeurs `x1` et `x2` s'écrit soit « `(x1, x2,)` » soit « `(x1, x2)` ».
- De manière générale, on accède au *n*-uplet constitué des valeurs de `x1`, `x2`, ..., `xn` (dans cet ordre) avec l'expression suivante : `(x1, x2, ..., xn,)`. Dans cette expression, la dernière virgule est facultative lorsqu'il y a au moins deux éléments.
- Les *n*-uplets sont immutables (on ne peut pas les altérer).
- Les *n*-uplets sont des itérables, il est donc possible de s'en servir pour construire des boucles *for*. Cela dit, il est plutôt rare que l'on ait besoin de le faire.
- Il est possible de concaténer deux *n*-uplets avec l'opérateur +. Cela dit, il est plutôt rare que l'on ait besoin de le faire.
- Les *n*-uplets sont surtout utilisés comme valeurs de retour de certaines fonctions. Par exemple :

```python
# x: int
def f(x):
  return ((x**2), (x**3))

print(f(2)) # Affiche "(4, 8)".
```

— Python ne contraint pas les éléments d'un *n*-uplet à être tous du même type. C'est quelque chose qui arrive très fréquement, notamment quand il s'agit de valeurs de retour de fonctions, et n'est pas considéré problématique. Par exemple :

```python
# x: int
def g(x):
  return ((x**2), f"x = {x}")

print(g(2)) # Affiche "(4, "x = 2")".
```

— Le nombre d'éléments d'un *n*-uplet est sa *longueur*. Comme pour une chaîne de caractères, on peut accéder à la longueur d'un *n*-uplet avec la fonction `len`. Par exemple, si `u=("Hello", "world", "!")`, `len(u)` vaut 3.

— On est rarement (bien que parfois) amené à utiliser la fonction `len` sur des *n*-uplet car ils sont surtout utilisés comme valeurs de retour de certaines fonctions retournant toujours des *n*-uplets de longueurs connues. Par exemple, la fonction `f` ci-dessus retourne toujours des paires (de longueur 2).

— Le système d'indexation des éléments d'un *n*-uplet est exactement le même que celui des caractères d'une chaîne de caractères. En particulier, si `u` est un *n*-uplet non vide, il est possible d'accéder à son premier élément avec `u[0]` et `u[-len(u)]`, et à son dernier élément avec `u[len(u)-1]` et `u[-1]`.

— Si l'on cherche à accéder aux différents éléments d'un *n*-uplet `u` de longueur k *connue au moment de la programmation*, on va souvent effectuer une assignation multiple des k éléments à k variables `x1`, `x2`, ..., `xk` avec l'instruction `x1, x2, ..., xk = u`. Par exemple :

```python
# x: int
def f(x):
  return ((x**2), (x**3))

a, b = f(3) # Assignation multiple.
print(a) # Affiche "9".
print(b) # Affiche "27".
```

Exercice 107 (Manipulation de paires, ⋆⋆)
Soit une fonction `f` prenant en argument un entier et renvoyant une paire de `float` · s, écrire une fonction `g` prenant en argument un entier `n` et renvoyant sous forme de paire les deux éléments de `f(n)` mais ordonnés dans l'ordre croissant.

□

5.2 Les listes

Le type `list` [Cours]

— Le type `list` est le type des structures de données appelées « listes » en Python. Une liste est une suite finie et ordonnée de valeurs appelées « éléments ».

— Comme nous le verrons par la suite, la différence majeure entre listes et *n*-uplets est que les listes sont mutables.

— Pour accéder à une liste constituée des valeurs de `x1`, `x2`, ..., `xn` (dans cet ordre), il est possible d'utiliser l'expression suivante, dite « en extension » : `[x1, x2, ..., xn]`. Par exemple :

```python
l = [39, 0, 0, 17, -29]
```

— La liste vide s'écrit `[]` ou, de manière équivalente, `list()`.
— Le nombre d'éléments d'une liste est sa *longueur*. Comme pour une chaîne de caractères, on peut accéder à la longueur d'une liste avec la fonction `len`. Par exemple, si `l=[8, 0, -4, 12, 0]`, `len(l)` vaut 5.
— Le système d'indexation des éléments d'une liste est exactement le même que celui des caractères d'une chaîne de caractères et des éléments d'un n-uplet. En particulier, si `l` est un liste non vide, il est possible d'accéder à son premier élément avec `l[0]` et `l[-len(l)]`, et à son dernier élément avec `l[len(l)-1]` et `l[-1]`.
— Une liste est un itérable. On peut donc s'en servir pour écrire une boucle *for*. Par exemple, le code suivant affiche chaque élément d'une liste sur une ligne (par élément), du premier au dernier :

```
l = [8, 0, -4, 12, 0]
for e in l:
  print(e)
```

— Pour visualiser une liste `l` dans son ensemble, utiliser `print(l)`. Pour visualiser seulement l'élément d'indice `i`, utiliser `print(l[i])`.
— Comme pour une chaîne de caractères ou un n-uplet, une *tranche* d'une liste `l` est une sous-liste contiguë de `l`. On accède à la tranche commençant à la position `i` (incluse) et se terminant à la position `j` (exclue) d'une liste `l` avec `l[i:j]`. Par exemple, si l'on a `l=[8, 0, -4, 12, 0]`, `l[1:3]` vaut `[0, -4]`, la tranche composée des éléments aux positions 1 et 2.
— Comme pour une chaîne de caractères, si $0 \leq$ `i` $<$ `j` $\leq$ `len(l)`, `len(l[i:j])` vaut toujours `j-i`.
— Comme pour une chaîne de caractères, `l[i:i]` vaut toujours `[]`. Si $0 \leq$ `i` $<$ `len(l)`, alors `l[i:(i+1)]` vaut `[l[i]]`. `l[0:len(l)]` vaut toujours `l`.
— Le mot-clef `in` permet de construire des expressions booléennes dénotant si un élement est contenu dans une liste. Par exemple, `(3 in [-4, 8, 3, 9])` vaut `True`.
— Python ne contraint pas les éléments d'une même liste à être tous du même type. Par exemple, `[1, 3.4, True, None, "coucou", [], [2]]` est une expression valide qui désigne une certaine liste de longueur 7. Cependant, il est une très bonne pratique d'éviter de manipuler de telles listes hétérogènes lorsque cela est possible.

Exercice 108 (Comprendre la taille et les indices, ⋆)
Supposons que `l1=[1, 3, 5, 7, 9, 11, 13, 15, 17]`.

1. *Que vaut* `len(l1)` *?*
2. *À quels indices (positifs et négatifs) trouve-t-on la valeur* 1 *?*
3. *À quels indices trouve-t-on la valeur* 17 *?*
4. *À quels indices trouve-t-on la valeur* 9 *?*

□

Exercice 109 (Comprendre la taille et les indices, ⋆)
Supposons que `l2=[2, 2, 3, 3, 4, 5, 7]`.

1. *Que vaut* `len(l2)` *?*
2. *Que vaut l'expression* `l2[0]` *?*
3. *Que vaut l'expression* `l2[4]` *?*
4. *Pourquoi l'évaluation de* `l2[7]` *génère-t-elle une erreur ?*

□

Exercice 110 (Parcourt à l'envers, ⋆)
Étant donnée une liste `l`*, écrire une boucle permettant d'afficher chaque élément de* `l` *sur une ligne (par élément), du dernier au premier.* □

Exercice 111 (Listes et expressions, ⋆⋆)
Supposons que `l=[1, 2, 4]`*. Que valent les expressions suivantes :*

1. `l[l[0]]`

2. `l[l[2] - l[1]]`

□

Exercice 112 (Listes d'itérables, ⋆)
Soit `l1=[[25, 10, 1917], [14, 7, 1789]]` *et* `l2=["Sabine", "Fred", "Jamy"]`.

1. *Que vaut* `l1[1]` *? Quel est son type ?*
2. *Que vaut* `l1[1][0]` *? Quel est son type ?*
3. *Que vaut* `l2[1]` *? Quel est son type ?*
4. *Que vaut* `l2[1][0]` *? Quel est son type ?*

□

Exercice 113 (Génération de phrases, ⋆⋆)
Soit `verbes=["parle avec", "voit"]` *et* `noms_propres=["Sabine", "Fred", "Jamy"]`.

1. *Écrire, à l'aide de boucles for, un code affichant toutes les phrases possibles constituées d'un sujet, d'un verbe et d'un objet à partir de* `verbes` *et* `noms_propres`.
2. *Modifier le code proposé à la question précédente pour ne générer que les phrases dont le sujet et l'objet sont distincts.*

□

Exercice 114 (Minimum et maximum, ⋆⋆⋆)

1. *Définir une fonction* `min` *prenant en arguments une liste* `l` *(qu'on supposera composée de valeurs numériques) et renvoyant le minimum de* `l` *si celle-ci est non vide et* `None` *sinon. La fonction doit être construite autour d'une boucle for.*
2. *Question similaire avec une fonction* `max` *renvoyant l'élément maximal de son argument s'il existe.*
3. *Définir une fonction* `min_and_max` *prenant en arguments une liste* `l` *(qu'on supposera composée de valeurs numériques) et renvoyant la paire composée du minimum et du maximum* `l` *si celle-ci est non vide et* `None` *sinon. La fonction ne doit pas faire appel aux deux fonctions* `min` *et* `max` *précédentes mais doit être construite autour d'une boucle for.*

□

Création en intension/compréhension [Cours]

— Étant donnée une liste `l=[x1, x2, ..., xn]` et une fonction à un argument `f`, il est possible d'accéder à la liste `[f(x1), f(x2), ..., f(xn)]` en utilisant l'expression suivante, dite « en intension » (ou « en compréhension ») : `[f(x) for x in l]`. Par exemple :

```
l1 = [1, -1, 2, -2, 3, -3]

# y: int
def f(y):
  return ((2 * y) + 1)

l2 = [f(x) for x in l1]
print(l2) # Affiche "[3, -1, 5, -3, 7, -5]".
```

— Il est à noter que bien qu'une expression en intension s'écrive avec le mot clef `for`, une expression en intension n'est pas une boucle *for*, ni ne contient, au sens propre, de boucle *for*. Il y a bien un lien conceptuel fort entre les deux concepts, mais ça n'en reste pas moins deux concepts distincts, correspondant à deux constructions syntaxiques distinctes.

— Plus généralement, on peut construire par intension une liste à partir d'une expression `exp`, d'un nom de variable `x` et d'un itérable de taille finie `s` (ex : une liste, une chaîne de caractères) avec l'expression

`[exp for x in s]`. Dans l'exemple suivant, l'expression `exp` est `(i**2)`, le nom de variable `x` est `i` et l'itérable `s` est `range(7)` :

```
l = [(i**2) for i in range(7)]
print(l) # Affiche "[0, 1, 4, 9, 16, 25, 36]".
```

— Le point précédent précise « itérable de taille *finie* » car il est possible en PYTHON de définir des itérables de taille infinie, mais nous n'en verrons aucun dans ce cours.

— La syntaxe précédente crée une liste contenant un élément pour chaque élément de l'itérable `s`. Il est possible d'intégrer une condition à une définition en intension pour ne générer des élements que pour certains éléments de `s`. Par exemple :

```
l1 = [(i**2) for i in range(7) if((i % 3) == 0)]
print(l1) # Affiche "[0, 9, 36]".

l2 = [(i**2) for i in range(7) if((i % 3) != 0)]
print(l2) # Affiche "[1, 4, 16, 25]".
```

Exercice 115 (Création par intension, ⋆)

1. *Définir par intension une liste* `l1` *valant* `[0, 2, 4, 6, 8, 10, 12, 14]`.
2. *Définir par intension une liste* `l2` *valant* `[2, 4, 6, 8, 10, 12, 14, 16]`.

□

Exercice 116 (Double ou triple, ⋆⋆)
Écrire une fonction `f` *prenant en argument une liste d'entiers* `l` *et renvoyant une liste de la même taille que* `l` *contenant à chaque indice* `i` *le double de* `l[i]` *si* `l[i]` *est positif et le triple sinon. La fonction* `f` *doit renvoyer une liste créée par intension à l'aide d'une fonction auxiliaire* `f_aux` *à définir.*

Contrat :
l = [1, -1, 2, -2] → *retour : [2, -3, 4, -6]*

□

Exercice 117 (Sommes d'entiers, ⋆⋆)
Écrire une fonction `f` *prenant en argument une liste d'entiers* `l` *supposés tous positifs et renvoyant une liste de la même taille que* `l` *contenant à chaque indice* `i` *la somme des entiers de 0 à* `l[i]`. *La fonction* `f` *doit renvoyer une liste créée par intension à l'aide d'une fonction auxiliaire* `f_aux` *à définir.*

Contrat :
l = [1, 4, 0, 3] → *retour : [1, 10, 0, 6]*

□

Exercice 118 (Pas de nombre pair, ⋆)
Écrire une fonction prenant en argument une liste d'entiers `l` *et retournant la liste des éléments de* `l` *qui ne sont pas pairs.* □

Exercice 119 (Pas d'espace, ⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et retournant la liste des caractères de* `s` *qui ne sont pas des espaces.* □

Exercice 120 (Pas de lettre en bas de casse, ⋆⋆)
Écrire une fonction prenant en argument une chaîne `s` *de caractères et retournant la liste des caractères de* `s` *qui ne sont pas des lettres en bas de casse.*

Contrat :
`s = "aBcD23"` → *retour :* `["B", "D", "2", "3"]`

□

5.3 Opérations sur les listes

Opérations fonctionnelles I [Cours]

— Il est possible de concaténer deux listes grâce à l'opérateur +. Par exemple, `([4, 3] + [1, 2])` est égale à `[4, 3, 1, 2]`.

— L'opérateur `*` permet répéter une liste, c.-à-d. de concaténer plusieurs copies de cette liste, pour former une nouvelle liste. Par exemple, `([1, 2] * 3)` est égale à `[1, 2, 1, 2, 1, 2]`.

— La fonction `enumerate` permet d'obtenir, à partir d'une liste `l`, un itérable associant à chaque élément de `l` sa position dans `l`. Soit `l=[x1, x2, ..., xn]`, `enumerate(l)` désigne une séquence d'éléments `(0, x1)`, `(1, x2)`, ..., `((n-1), xn)` ; ses éléments sont donc des paires de valeurs, composées d'un entier et d'un élément de `l`. Par exemple :

```
l = [-2, 9, 0, 0, 25, 3]
for (i, x) in enumerate(l):
  print(f"The element at position {i} is {x}.")
```

— La fonction `zip` permet de combiner plusieurs itérables en un itérable de n-uplets. Les itérables fournis en argument sont parcourus de manière synchrone jusqu'à ce que l'un d'entre eux soit entièrement parcouru. Par exemple, le code suivant affiche successivement les lignes « b 0 mot1 », « o 1 mot2 », « n 2 mot3 » et « j 3 mot4 » :

```
l = ["mot1", "mot2", "mot3", "mot4", "mot5"]
for x, y, z in zip("bonjour", range(4), l):
  print(x, y, z)
```

— La fonction `reversed` permet d'obtenir, à partir d'une liste `l`, une séquence contenant les éléments de `l` dans l'ordre inverse. Par exemple, le code suivant affiche « 3 25 0 0 9 -2 » :

```
l = [-2, 9, 0, 0, 25, 3]
for x in reversed(l):
  print(x, end=" ")
```

— Les fonctions `enumerate`, `zip` et `reversed` retournent des séquences qui ne sont pas du type `list` :

```
l = [1, -1, 2, -2]
print(enumerate(l)) # Affiche "<enumerate object at 0
    x7fd117371200>".
print(zip(l, l)) # Affiche "<zip object at 0x7ffa91c84140>".
print(reversed(l)) # Affiche "<list_reverseiterator object at
    0x7fd11640d760>".
```

— Tout itérable peut cependant être converti en une liste à l'aide de la fonction `list`, à condition que l'itérable soit de taille finie (ce qui est toujours le cas pour des séquences obtenues par `enumerate`, `zip` et `reversed` sur des listes) :

```
l = [1, -1, 2, -2]
print(list(enumerate(l))) # Affiche "[(0, 1), (1, -1), (2, 2),
    (3, -2)]".
print(list(zip(l, l))) # Affiche "[(1, 1), (-1, -1), (2, 2),
    (-2, -2)]".
print(list(reversed(l))) # Affiche "[-2, 2, -1, 1]".
```

— Une chaîne de caractères est une séquence de longueur finie et peut donc être convertie en une liste de caractères à l'aide de la fonction `list` :

```
s = 'Coucou'
print(list(s)) # Affiche "['C', 'o', 'u', 'c', 'o', 'u']".
```

— Il existe une fonction très similaire à `list` : `tuple`, qui convertit tout itérable de taille finie en un n-uplet. Par exemple :

```
print(tuple("Coucou")) # Affiche "('C', 'o', 'u', 'c', 'o', 'u')".
print(tuple([0, 1, 2])) # Affiche "(0, 1, 2)".
```

Exercice 121 (Sandwich, ⋆)
Écrire une fonction prenant en argument deux listes `l1` *et* `l2` *et renvoyant la liste composée des éléments de* `l1`*, puis de* `l2` *puis encore de* `l1`*.*

Contrat :
l1, l2 = [1, 2], [8, 9, 0] → *retour : [1, 2, 8, 9, 0, 1, 2]*

□

Exercice 122 (Lettres à répétition, ⋆)

1. *Écrire une fonction prenant en argument un entier* `n` *et une chaîne de caractères* `s`*, et renvoyant la liste des caractères de* `s` *répétée* `n` *fois.*

   Contrat :
   n, s = 3, "az" → *retour : ["a", "z", "a", "z", "a", "z"]*
   n, s = 2, "c" → *retour : ["c", "c"]*
2. *Proposer une autre manière (très proche) d'écrire cette fonction.*

□

Exercice 123 (Taille fixée, ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et une liste* `l`*, et retournant une liste de taille* `n` *remplie par les éléments de* `l` *à partir du premier et éventuellement complétée par des 0.*

Contrat :
n, l = 3, [8, 2, 3, 9, 9] → *retour : [8, 2, 3]*
n, l = 7, [8, 2, 3, 9, 9] → *retour : [8, 2, 3, 9, 9, 0, 0]*

□

Exercice 124 (Zip, ⋆⋆)

1. *Soit* `l = ["une", "liste", "de", "mots"]`*, que vaut* `list(zip(range(len(l)), l))` *?*
2. *Quelle expression plus simple a exactement la même valeur ?*

□

Opérations fonctionnelles II [Cours]

— La fonction `sorted` permet d'obtenir, à partir d'une liste `l`, une séquence contenant les éléments de `l` triés par ordre croissant en utilisant l'opérateur > (qui désigne notamment l'ordre usuel sur les types numériques et une variante de l'ordre lexicographique pour les chaînes de caractères). Par exemple :

```
l1 = [1, -1, 2, -2]
print(sorted(l1)) # Affiche "[-2, -1, 1, 2]".

l2 = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l2)) # Affiche "['aliment', 'aluminium', 'renard', 'université', 'zèbre']".
```

— Par défaut, la sortie de `sorted` est triée dans l'ordre croissant, mais `sorted` peut aussi produire une séquence d'éléments triés dans l'ordre décroissant si l'on spécifie son argument spécial `reverse` à `True`. Par exemple :

```
l = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l, reverse=True)) # Affiche "['zèbre', 'universit
    é', 'renard', 'aluminium', 'aliment']".
```

— Par défaut, la sortie de `sorted` est triée suivant l'ordre défini par l'opérateur > sur *les élements* de l'itérable d'entrée *eux-mêmes*, mais `sorted` peut aussi produire une séquence triée suivant l'ordre défini par > sur *une propriété* des éléments de l'itérable. Pour ce faire, il suffit de spécifier l'argument spécial `key` de `sorted` en indiquant une fonction renvoyant la propriété qui doit être utilisée pour le tri. Par exemple, le code suivant trie les chaînes de `l` en fonction de leur dernière lettre :

```
# s: str
def get_last_char(s):
  if(len(s) >= 1): return s[-1]
  return ""

l = ["zèbre", "aliment", "aluminium", "université", "renard"]
print(sorted(l, key=get_last_char)) # Affiche "['renard', 'zè
    bre', 'aluminium', 'aliment', 'université']".
```

— Toutes les opérations vues dans ce bloc de cours et le précédent sont *fonctionnelles*, c.-à-d. qu'elles génèrent de nouvelles listes ou séquences sans modifier la liste initiale. Par exemple, après exécution de la suite d'instructions suivante, seule la liste `l2` est triée, pas `l1` :

```
l1 = [1, -1, 2, -2]
l2 = sorted(l1)
print(l2) # Affiche "[-2, -1, 1, 2]".
print(l1) # Affiche "[1, -1, 2, -2]".
```

— Les fonctions `enumerate`, `reversed` et `sorted` peuvent prendre tout type d'itérable comme argument et donc en particulier toute chaîne de caractères :

```
s = "Coucou"
print(list(enumerate(s))) # Affiche "[(0, 'C'), (1, 'o'), (2,
    'u'), (3, 'c'), (4, 'o'), (5, 'u')]".
print(list(reversed(s))) # Affiche "['u', 'o', 'c', 'u', 'o',
    'C']".
print(sorted(s)) # Affiche "['C', 'c', 'o', 'o', 'u', 'u']".
```

Exercice 125 (Trier en fonction de la longueur, ⋆⋆)
Écrire une fonction prenant en argument une liste de chaînes de caractères `l` *et renvoyant la liste des éléments de* `l` *triés en fonction de leur longueur par ordre croissant.*

Contrat :
l =["aa", "cba", "b"] → *retour : ["b", "aa", "cba"]*

□

Exercice 126 (Trier des entiers par ordre décroissant, ⋆⋆)
Étant donnée une liste d'entiers `l`*, donner trois manières différentes pour obtenir la liste des éléments de* `l` *triés dans l'ordre décroissant.* □

Opérations non fonctionnelles I [Cours]

— Contrairement aux chaînes de caractères ou aux *n*-uplets, les listes sont *mutables*, c.-à-d. altérables : leurs éléments et le nombre de ces éléments sont modifiables.

— Soit une liste `l`, une position `i` définie dans `l` et `x` une valeur, l'instruction d'assignation `l[i] = x` remplace l'élément d'indice `i` de `l` par `x`. Par exemple :

```
l = [9, 8, 2, 8]
l[1] = 0

print(l) # Affiche "[9, 0, 2, 8]".
```

— L'opérateur `*` permet par exemple d'initialiser une liste avant de la remplir avec les valeurs voulues, grâce à des assignations, lorsque l'on connaît à l'avance le nombre d'éléments. Par exemple :

```
l = [0] * 10
l[3] = 2
l[7] = 1

print(l) # Affiche "[0, 0, 0, 2, 0, 0, 0, 1, 0, 0]".
```

— La méthode `append` permet de rajouter un élément à la fin d'une liste, augmentant ainsi sa longueur d'une unité. `append` retourne toujours `None` mais modifie la liste elle-même. Par exemple :

```
l = []
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=0; l=[]".

l.append(2)
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=1; l=[2]".

l.append(-7)
print(f"len(l)={len(l)}; l={l}") # Affiche "len(l)=2; l=[2,
    -7]".
```

— La méthode `pop` est en quelque sorte l'inverse d'`append` : appelée sur une liste non vide, elle supprime le dernier élément de la liste et renvoie cette valeur. Attention, si `pop` est appelée sur une liste vide, une exception est levée. Par exemple :

```
l = list(range(3)) # [0, 1, 2]

print(l.pop()) # Affiche "2"
print(l) # Affiche "[0, 1]"

print(l.pop()) # Affiche "1"
print(l) # Affiche "[0]"

print(l.pop()) # Affiche "0"
print(l) # Affiche "[]"

print(l.pop()) # IndexError: pop from empty list
```

— La méthode `extend` permet d'étendre une liste avec tous les éléments d'une seconde liste. Par exemple :

```
l = [0, 1, 2, 3]

l.extend([4, 5, 6])
print(l) # Affiche "[0, 1, 2, 3, 4, 5, 6]".

l.extend([])
print(l) # Affiche "[0, 1, 2, 3, 4, 5, 6]".
```

— L'argument de la méthode `extend` peut être n'importe quel itérable de taille finie. Par exemple :

```
l = []

l.extend(range(3))
print(l) # Affiche "[0, 1, 2]".
```

— Attention à ne pas confondre `extend` et `append`. Bien que l'utilisation de la méthode `extend` avec pour argument une valeur qui n'est pas un itérable génère une erreur, il est à l'inverse tout à fait possible de donner un itérable comme argument à `append` ; simplement, c'est l'itérable lui-même qui sera ajouté en tant qu'élément (un seul, quelle que soit sa taille) à la liste.

```
l = [0, 1, 2]

l.extend(3) # Erreur

l.append([3, 4])
print(l) # Affiche "[0, 1, 2, [3, 4]]".
print(l == [0, 1, 2, 3, 4]) # Affiche "False".
```

Exercice 127 (Inversion d'éléments, ⋆⋆)
Considérer une liste `l` *quelconque.*

1. *Écrire une suite d'instructions remplaçant, si possible, le troisième élément de* `l` *par la valeur* `1`. *(Veiller à ce que le code s'éxecute sans erreur dans tous les cas.)*
2. *Écrire une suite d'instructions inversant, si possible, le premier et le second élément de* `l`.
3. *Écrire une suite d'instructions inversant, si possible, le premier et le dernier élément de* `l`.

□

Exercice 128 (Écriture, ⋆)
Après exécution de la suite d'instructions suivante, que vaut `l` *?*

```
l = [2, 3, 4, 5, 6, 7]
for i in range(len(l)):
  l[i] = i
```

□

Exercice 129 (Définition par intension et construction itérative, ⋆⋆)

1. *Écrire une suite d'instructions construisant la liste* `[(i**2) for i in range(7)]` *sans utiliser de définitions par intension mais à l'aide notamment d'une boucle for et de la méthode* `append`.
2. *Même question pour la liste* `[(i**2) for i in range(7) if((i % 3) == 0)]`.

□

Exercice 130 (extend et range, ⋆)
Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?

```
l = []

l.extend(range(3))
print(l)

l.extend(range(0))
print(l)

l.extend(range(1))
print(l)

l.extend(range(2))
print(l)
```

□

Exercice 131 (Doubler chaque élément, ⋆⋆)
Écrire une fonction prenant en argument une liste `l` *et renvoyant la liste composée de chaque élément de* `l` *répété deux fois de suite.*

Contrat :
l = [8, 9, 0] → *retour : [8, 8, 9, 9, 0, 0]*

□

Exercice 132 (Intensification, ⋆⋆)
Écrire une fonction prenant en argument une liste `l` *de chaînes de caractères et renvoyant la liste composée de chaque élément de* `l` *mais en répétant deux fois de suite toute occurrence de* `"très"`.

Contrat :
l = ["Un", "très", "grand", "arbre", "."] → *retour : ["Un", "très", "très", "grand", "arbre", "."]*

□

Opérations non fonctionnelles II [Cours]

— La méthode `reverse` inverse la liste sur laquelle elle est appelée. Par exemple :

```
l = [-2, 9, 0, 0, 25, 3]

l.reverse()
print(l) # Affiche "[3, 25, 0, 0, 9, -2]".
```

— La méthode `sort` trie la liste sur laquelle elle est appelée. Cette méthode accepte les mêmes arguments spéciaux `reverse` et `key` que la fonction `sorted`. Par exemple :

```
l = ["zèbre", "aliment", "aluminium", "université", "renard"]

l.sort()
print(l) # Affiche "['aliment', 'aluminium', 'renard', '
    université', 'zèbre']".

l.sort(reverse=True, key=get_last_char)
print(l) # Affiche "['université', 'aliment', 'aluminium', 'zè
    bre', 'renard']".
```

— Les méthodes `append`, `extend`, `pop`, `sort` et `reverse` ne sont pas fonctionnelles. Toutes modifient

la liste sur laquelle elles sont appelées et retournent toujours `None`.

— Il est important de noter que si l'on donne en argument à une fonction une valeur mutable (ex : une liste) et que l'on modifie cette valeur (ex : avec une opération non fonctionnelle) dans le corps de la fonction, alors la modification persiste après la fin de l'exécution de la fonction :

```
#l: list[int]
def f(l):
  l.reverse() # Opération non fonctionnelle.

l1 = [0, 1, 2, 3]
f(l1)
print(l1) # Affiche "[3, 2, 1, 0]".
```

Durant l'exécution de l'exemple précédant, la variable locale `l` (argument de `f`) ne désigne pas une copie de la liste désignée par `l1` mais cette liste directement. C'est pourquoi l'appel à la méthode `reverse` dans le corps de `f` a un effet observable même après la fin de l'exécution de `f` (ce que l'on appelle un « effet de bord ») : il n'y a dans cet exemple qu'une seule liste en mémoire, désignée à la fois par `l1` et `l`, et qui est inversée dans le corps de la fonction `f`.

Exercice 133 (Opérations fonctionnelles et non fonctionnelles, ⋆⋆)

1. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f1(l, x):
  return l.append(x)

l = [-1, 0, 1, 2]
print(f1(l, 3))
print(l)
```

2. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f2(l, x):
  return l + [x]

l = [-1, 0, 1, 2]
print(f2(l, 3))
print(l)
```

3. *Quel est l'affichage produit par l'exécution de la suite d'instructions suivante ?*

```
# l: list
# x: any
def f3(l, x):
  l.append(x)
  return l

l = [-1, 0, 1, 2]
print(f3(l, 3))
print(l)
```

□

Copie ou non [Cours]

— Nous avons déjà mentionné plus haut la fonction `list`, qui sert essentiellement à convertir les itérables de taille finie en listes. Si on lui passe en argument une liste `l`, elle renvoie une *copie* de `l` :

```
l1 = [-1, 1, -2, 2]
l2 = list(l1)
l1.extend([-3, 3])
print(l1) # Affiche "[-1, 1, -2, 2, -3, 3]".
print(l2) # Affiche "[-1, 1, -2, 2]".
```

— Dans un certain nombre de cas, les mêmes opérations effectuées avec `+`, `append` ou `extend` peuvent être effectuées avec une autre de ces fonctions. Par exemple, les valeurs de `l1` et `l2` sont les mêmes après exécution de n'importe laquelle des trois suites d'instructions suivantes :

```
l1 = [0, 1, 2]
l2 = [3, 4]
l1 = l1 + l2
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

```
l1 = [0, 1, 2]
l2 = [3, 4]
l1.extend(l2)
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

```
l1 = [0, 1, 2]
l2 = [3, 4]
for x in l2: l1.append(x)
print(l1, l2) # Affiche "[0, 1, 2, 3, 4] [3, 4]".
```

— Ces différentes opérations fonctionnent cependant de manière radicalement différentes. (Les descriptions qui suivent ignorent un certain nombre de détails techniques qui sont cependant sans impact sur les conclusions.)
  — `l1 = l1 + l2` : cette instruction (i) commande l'allocation en mémoire d'une nouvelle liste de taille `(len(l1) + len(l2))`, (ii) y copie le contenu de `l1` à partir de la position `0` puis (iii) le contenu de `l2` à partir de la position `len(l1)`, et enfin (iv) assigne cette nouvelle liste au nom de variable `l1`. Le *coût* de cette instruction (le temps requis pour son exécution) est donc grosso modo proportionnel à la somme des longueurs de `l1` et `l2`.
  — `l1.extend(l2)` : cette instruction se contente d'étendre la liste `l1` de `len(l2)` cases et d'y copier le contenu de `l2`. Le coût de cette instruction est donc proportionnel à la longueur de `l2`.
  — `for x in l2: l1.append(x)` : pour chaque élément de `l2`, cette instruction étend `l1` d'une case et y copie cet élément. Le coût de cette instruction est donc proportionnel à la longueur de `l2`.

— La différence de coût entre une concaténation (`+`) et une extension (`append`, `extend`) de liste peut être problématique, notamment si l'opération est répétée, comme lorsque l'on utilise une liste comme accumulateur. Il y a par exemple un facteur ≈ 500 entre les coûts des deux suites d'instructions suivantes :

```
l = []
for i in range(1000):
  l += [(3 * i) + 1] # Complexité : 1, puis 2, 3, ..., 1000.
```

```
l = []
for i in range(1000):
  l.append((3 * i) + 1) # Complexité : 1, puis 1, 1, ..., 1.
```

— Il est courant de construire des listes par acculumation ; il faut dans ces cas-là utiliser `append` ou `extend` et non `+`.

— Si l'on cherche à *créer* une nouvelle liste, on utilisera l'opérateur +. Si l'on cherche à *modifier* une liste, on utilisera les méthodes `append` ou `extend`.
— Les chaînes de caractères n'étant pas mutables, il n'y a pas d'équivalent des méthodes `append` ou `extend` pour le type `str`. Le coût de la concaténation sur les chaînes de caractères étant similaire à celui sur les listes, si l'on souhaite construire une chaîne de caractères par accumulation, le mieux est de créer par accumulation une liste de chaînes de caractères puis de les concaténer « en bloc » avec la méthode `join`. Par exemple :

```
l = []
for i in range(100):
  l.append(f"coucou {i}")
s = "".join(l) # Complexité : somme des longueurs des chaînes
   de `l`.
```

— La méthode `join` a pour argument la liste de chaînes de caractères à joindre et s'appelle sur une autre chaîne de caractères, insérée entre les premières dans le résultat. Par exemple :

```
l = list("abcde") # ['a', 'b', 'c', 'd', 'e']
print("--".join(l)) # Affiche "a--b--c--d--e".
print(" ".join(l)) # Affiche "a b c d e".
print("".join(l)) # Affiche "abcde".
```

Exercice 134 (Concaténer les éléments d'une liste, ⋆⋆)

1. *En utilisant* `join`*, écrire une fonction prenant en argument une liste* `names` *de chaînes de caractères, et renvoyant la chaîne obtenue par la concaténation des éléments de* `names`*, séparées par « , » (un point et un espace).*

   Contrat :
   names = *["Sabine", "Fred", "Jamy"]* → *retour : "Sabine, Fred, Jamy"*
   names = *[]* → *retour : ""*

2. *Même question mais sans utiliser* `join`*.*
3. *Parmi les deux fonctions proposées, laquelle est préférable en termes de complexité ?*

□

Exercice 135 (Concaténer les éléments d'une liste différement, ⋆⋆)

1. *Sans utiliser* `join`*, écrire une fonction prenant en argument une liste* `names` *de chaînes de caractères, et renvoyant la chaîne obtenue par la concaténation des éléments de* `names` *séparées par « , » (un point et un espace) sauf pour les deux dernières, qui doivent être séparées par « et ».*

   Contrat :
   names = *["Sabine", "Fred", "Jamy"]* → *retour : "Sabine, Fred et Jamy"*
   names = *["Sabine"]* → *retour : "Sabine"*
   names = *[]* → *retour : ""*

2. *Même question mais en utilisant* `join`*.*
3. *Parmi les deux fonctions proposées, laquelle est préférable en termes de complexité ?*

□

5.4 Exercices supplémentaires

Exercice 136 (Multiplier chaque élément, ⋆⋆)
Écrire une fonction prenant en argument un entier `n` *et une liste* `l`*, et renvoyant la liste composée de chaque élément de* `l` *répété* `n` *fois de suite.*

Contrat :
n, l = 3, [8, 9, 0, 1] → *retour : [8, 8, 8, 9, 9, 9, 0, 0, 0, 1, 1, 1]*
n, l = 0, [8, 9, 0, 1] → *retour : []*

□

Exercice 137 (Somme, ⋆)
Écrire une fonction `sum_list` *prenant en argument une liste de valeurs numériques, et renvoyant la somme de ces valeurs (et en particulier 0 si la liste est vide). La fonction doit être construite autour d'une boucle for.* □

Exercice 138 (Moyenne, ⋆)
Écrire fonction `mean_list` *prenant en argument une liste de valeurs numériques, et renvoyant la moyenne de ces valeurs ou* `None` *si la liste est vide.* □

Exercice 139 (Grelottement, ⋆)
Écrire une fonction prenant en argument un entier `n` *et affichant les* `n` *premiers « grelottements » : « brhh », « brrhh », « brrrhh », etc.* □

Exercice 140 (Première occurrence, ⋆⋆)
Écrire une fonction `first_occ` *prenant en argument une liste d'entiers* `l` *et un entier* `n`*, et renvoyant l'indice de la première occurrence de* `n` *dans* `l` *ou -1 s'il n'existe aucune telle occurrence.* □

Exercice 141 (Dernière occurrence, ⋆⋆)
Écrire une fonction `last_occ` *prenant en argument une liste d'entiers* `l` *et un entier* `n`*, et renvoyant l'indice de la dernière occurrence de* `n` *dans* `l` *ou -1 s'il n'existe aucune telle occurrence.* □

Exercice 142 (Suite dans une liste, ⋆⋆)
Écrire une fonction `progression` *prenant en argument trois entiers* `a`, `b` *et* `n`*, et renvoyant la liste* `[a, (a + b), (a + 2*b), (a + 3*b), ..., (a + (n-1)*b)]`. □

Exercice 143 (Énumération, ⋆⋆)
Écrire une fonction prenant en argument un itérable de taille finie `s` *et renvoyant la liste associant à chaque élément de* `s` *sa position dans* `s`*. La fonction* `enumerate` *ne doit pas être utilisée.*

Contrat :
s = [8, 9, 0] → *retour : [(0, 8), (1, 9), (2, 0)]*
s = "Hello" → *retour : [(0, "H"), (1, "e"), (2, "l"), (3, "l"), (4, "o")]*

□

Exercice 144 (Entrelacement, ⋆⋆)
Écrire une fonction `interlace` *prenant en argument deux listes* `l1` *et* `l2` *supposées de même longueur, et renvoyant une liste de longueur double qui contient les valeurs des deux listes de façon entrelacée, c.-à-d.* `[l1[0], l2[0], l1[1], l2[1], ..., l1[len(l1)], l2[len(l1)]]`.

Contrat :
l1, l2 = [0, 1, 6], [2, 4, 7] → *retour : [0, 2, 1, 4, 6, 7]*

□

Exercice 145 (Plagiat, ⋆⋆)

1. *Écrire une fonction* `plagiarism` *prenant en argument deux listes* `l1` *et* `l2`*, et renvoyant une paire d'indices* `(i, j)` *telle que* `l1[i] == l2[j]` *si une telle paire existe et* `None` *sinon.*
2. *Écrire une fonction* `auto_plagiarism` *prenant en argument une liste* `l` *et renvoyant une paire d'indices* `(i, j)` *telle que* `(l[i] == l[j]) and (i < j)` *si une telle paire existe et* `None` *sinon.*

□

Exercice 146 (Fonctions et liste de chaînes de caractères, ⋆⋆)

1. *Que fait la fonction* `func_ab` *définie de la manière suivante ?*

```python
# n: int
def func_ab(n):
    l = []
    s = "ab"
    for _ in range(n):
        l.append(s)
        s = s + "ab"

    return l
```

2. *Simplifier cette fonction à l'aide d'une expression par intension.*
3. *Comparer la complexité de ces deux implémentations en fonction de la valeur de l'argument* `n`.

□

Exercice 147 (Comptage, ⋆⋆)

Écrire une fonction `comptage` *prenant en argument une liste d'entiers* `l` *et un nombre entier* `n`*, et renvoyant la liste d'entiers dont l'élément d'indice* `i` *(entre 0 et* `n` *inclus) est le nombre d'occurrences de l'entier* `i` *dans* `l`*.*

Contrat :
`l, n = [0, 1, 2, 2, 0, 0], 4` → *retour :* `[3, 1, 2, 0, 0]`
`l, n = [0, 1, 2, 2, 0, 0], 1` → *retour :* `[3, 1]`

□

Chapitre 6

Débogage

Objectifs :

- Savoir détecter les erreurs dans du code et le déboguer.
- Savoir implémenter un certain nombre d'opérations classiques sur les listes.

6.1 Implémentation d'opérations classiques sur les listes : débogage

Exercice 148 (Présence dans une liste, ⋆)
On cherche à implémenter une fonction `is_present` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant* `True` *si l'un des éléments de* `l` *vaut* `x` *et* `False` *sinon. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat erroné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```
# x: any; l: list
def is_present(x, l):
  for y in l:
    if(y == x):
      return True
    else:
      return False

  return False
```

2.
```
# x: any; l: list
def is_present(x, l):
  presence = True
  for y in l:
    if(y == x):
      presence = True
    else:
      presence = False

  return presence
```

3.
```python
# x: any; l: list
def is_present(x, l):
  presence = False
  i = 0
  while((i < len(l)) and presence):
    if(l[i] == x):
      presence = True

  return presence
```

4.
```python
# x: any; l: list
def is_present(x, l):
  presence = False
  i = 0
  while((i < len(l)) and (not presence)):
    presence = (presence and (l[i] == x))

  return presence
```

□

Exercice 149 (Première position dans une liste, ⋆)
On cherche à implémenter une fonction `first_pos` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant le premier indice dans* `l` *d'un élément valant* `x` *si un tel indice existe et* `None` *sinon. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat eronné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```python
# x: any; l: list
def first_pos(x, l):
  for i in range(len(l)):
    if(x == l[i]):
      return i
    else:
      return None

  return None
```

2.
```python
# x: any; l: list
def first_pos(x, l):
  p = 0
  for i in range(len(l)):
    if(x == l[i]):
      p = i
    else:
      p = None

  return p
```

3.
```python
# x: any; l: list
def first_pos(x, l):
  p = None
  i = 0
  while(i < len(l)):
    if(x == l[i]):
      p = i
      i += 1

  return p
```

4.
```python
# x: any; l: list
def first_pos(x, l):
  p = None
  i = 0
  while((i < len(l)) and (p != None)):
    if(x == l[i]):
      p = i
    i += 1

  return p
```

□

Exercice 150 (Compter dans une liste, ⋆)

On cherche à implémenter une fonction `count` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant le nombre d'occurrences de* `x` *dans* `l`*. Pour chacune des propositions suivantes,*

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat eronné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```python
# x: any; l: list
def count(x, l):
  k = 0
  for y in l:
    if(x == y):
      k += 1
      return k

  return k
```

2.
```python
# x: any; l: list
def count(x, l):
  k = len(l)
  for y in l:
    if(x == y):
      k += 1
    else :
      k -= 1

  return k
```

□

Exercice 151 (Construire la liste des positions, ⋆)
On cherche à implémenter une fonction `pos` *prenant en argument une valeur quelconque* `x` *et une liste* `l`*, et renvoyant la liste des indices dans* `l` *où* `x` *apparaît.*

Contrat :

`x, l = 2, [3, 4, 5, 8]`	→	*retourne :* `[]`
`x, l = 3, [3, 3, 4, 3, 4, 5, 6, 4, 3]`	→	*retourne :* `[0, 1, 3, 8]`
`x, l = 3, [3, 3, 4, 3, 4, 5, 6, 4, 3]`	→	*retourne :* `[2, 4, 7]`

Pour chacune des propositions suivantes,

- *expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes) ;*
- *donner un exemple de valeurs de* `x` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat erroné) ;*
- *effectuer une correction minimale de cette proposition.*

1.
```
# x: any; l: list
def pos(x, l):
  res = []
  for i in range(len(l)):
    if(l[i] == x):
      res = [i]

  return res
```

2.
```
# x: any; l: list
def pos(x, l):
  return [i for i in range(l) if(x == i)]
```

3.
```
# x: any; l: list
def pos(x, l):
  return [l[i] for (i, y) in enumerate(l) if(x == y)]
```

□

Exercice 152 (Échanger deux éléments dans une liste, ⋆)
On cherche à implémenter une fonction `swap` *prenant en argument deux entiers* `i` *et* `j` *et une liste* `l`*, et échangeant les éléments aux indices* `i` *et* `j` *dans* `l`*. Considérer la proposition suivante.*

- *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
- *Donner un exemple de valeurs de* `i`*,* `j` *et* `l` *pour lesquelles la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
- *Effectuer une correction minimale de cette proposition.*

```
# i, j: int; l: list
def swap(i, j, l):
  l[i] = l[j]
  l[j] = l[i]
```

□

Exercice 153 (Inverser une liste, ⋆⋆)
On cherche à implémenter une fonction `my_reverse` *prenant en argument une liste* `l` *et inversant* `l` *comme le ferait* `l.reverse`*. Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `my_reverse(l)`*,* `l` *vaudra* `['d', 'c', 'b', 'a']`*. Considérer la proposition suivante.*

— *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def reverse(l):
  for i in range(len(l)):
    l[i] = l[len(li) - 1 - i]
```

□

Exercice 154 (Rotation avant dans une liste, ⋆⋆⋆)
On cherche à implémenter une fonction `rotate_r` *prenant en argument une liste* `l` *et décalant tous les éléments de* `l` *vers l'indice supérieur (ou au début de la liste pour le dernier élément). Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `rotate_r(l)`*,* `l` *vaudra* `['d', 'a', 'b', 'c']`*. Considérer la proposition suivante.*

— *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def rotate_r(l):
  if(len(l) < 1): return # Equivalent à "return None".

  tmp = l[-1]
  for i in range(len(l), 0, -1): l[i] = l[i-1]
  l[0] = tmp
```

□

Exercice 155 (Rotation arrière dans une liste, ⋆⋆⋆)
On cherche à implémenter une fonction `rotate_l` *prenant en argument une liste* `l` *et décalant tous les éléments de* `l` *vers l'indice inférieur (ou à la fin de la liste pour le premier élément). Par exemple, si* `l=['a', 'b', 'c', 'd']`*, juste après l'exécution de* `rotate_l(l)`*,* `l` *vaudra* `['b', 'c', 'd', 'a']`*. Considérer la proposition suivante.*

— *Expliquer pourquoi celle-ci est fausse (il peut y avoir plusieurs problèmes).*
— *Donner un exemple de valeur de* `l` *pour laquelle la fonction ne se comporte pas comme souhaité (indiquer le résultat éronné).*
— *Effectuer une correction minimale de cette proposition.*

```
# l: list
def rotate_l(l):
  if(len(l) < 1): return # Equivalent à "return None".

  tmp = l[0]
  for i, x in enumerate(l): l[i-1] = x
  l[-1] = tmp
```

□

6.2 Exercices supplémentaires

Exercice 156 (Addition, ⋆⋆⋆)
Dans cet exercice, les nombres entiers sont représentés par des listes de chiffres. Plus précisément, un entier est représenté par une liste de valeurs de type `int` *toutes entre 0 et 9 (inclus) et se lisant de la droite vers la gauche, c.-à-d. que le premier élément de la liste est le nombre d'unités, le second élément est le nombre de dizaines, le troisième élément est le nombre de centaines, etc. Par exemple,* 843 *est représenté par* `[3, 4, 8]` *et* 29 *par* `[9, 2]`.

Écrire une fonction `add` *prenant en argument deux listes* `l1` *et* `l2`*, et retournant la liste représentant la somme des deux nombres représentés par* `l1` *et* `l2`.

Par exemple, si les entrées sont les listes `l1=[3, 4, 8]` *et* `l2 = [9, 2]`*, la valeur à retourner est la liste* `[2, 7, 8]`*, qui représente le nombre* 872*. Le premier élément,* `2`*, s'obtient en additionnant* `3` *et* `9` *et en notant la retenue. Le second élément,* `7`*, s'obtient en additionnant* `4`*,* `2` *et la retenue précédente. Le troisième élément,* `8`*, s'obtient directement à partir du* `8`*, le nombre de centaines du premier nombre. Le calcul s'arrête là car il n'y a plus de chiffre à additionner ni de retenue.*

Contrat :
`l1, l2 = [5, 3, 4, 2], [6, 3, 4]` → *retour :* `[1, 7, 8, 2]`
`l1, l2 = [2, 1], [3, 1]` → *retour :* `[5, 2]`
`l1, l2 = [1], [9, 9]` → *retour :* `[0, 0, 1]`

□

Chapitre 7

Ensembles et dictionnaires

Objectifs :

- Définir des ensembles en extension et en intension/compréhension.
- Combiner des ensembles.
- Modifier des ensembles.
- Déclarer des dictionnaires en extension et en intension/compréhension.
- Parcourir les clefs et/ou les valeurs d'un dictionnaire.
- Modifier des dictionnaires.

7.1 Les ensembles

Le type `set` [Cours]

- Le type `set` est le type des structures de données appelées « ensemble ». Un ensemble est une structure mutable représentant une collection *non ordonnée* de valeurs *distinctes* appelées « éléments ».
- Les éléments d'un ensemble ne peuvent pas être de n'importe quel type. Parmi les types natifs, les types non mutables (ex : `int`, `str`, `tuple`) peuvent être utilisés pour les éléments d'un ensemble, mais pas les types mutables (ex : `list`, `set`).
- Pour accéder à un ensemble contenant les éléments (au moins un) `x1`, `x2`, ..., `xn`, il est possible d'utiliser l'expression suivante, dite « en extension » : `{x1, x2, ..., xn}`. Par exemple :

```
s = {"Sabine", "Fred", "Jamy"}
```

- Attention, l'expression `{}` ne vaut pas l'ensemble vide (mais le *dictionnaire* vide ; les dictionnaires sont étudiés plus bas). On peut par contre désigner l'ensemble vide avec l'expression `set()`.
- La fonction `print` appelée sur un ensemble affiche ses éléments dans un ordre arbitraire (c.-à-d. pouvant varier d'une exécution à l'autre).
- Le nombre d'éléments d'un ensemble est sa *longueur*. Comme pour une chaîne de caractères ou une liste, on peut accéder à la longueur d'un ensemble avec la fonction `len`. Par exemple, `len({"Sabine", "Fred", "Jamy"})` vaut 3.
- Par définition, les éléments d'un ensemble sont tous distincts et donc, si une valeur apparaît plusieurs fois dans la définition d'un ensemble, elle n'apparaîtra quand même qu'une seule fois dans l'ensemble. Par exemple :

```
s1 = {"Sabine", "Fred", "Jamy"}
s2 = {"Sabine", "Fred", "Jamy", "Sabine"}
print(s2) # Affiche "{'Fred', 'Sabine', 'Jamy'}".
print(s1 == s2) # Affiche "True".
```

- Par définition, un ensemble est non ordonné. L'éventuel ordre des valeurs dans la définition d'un ensemble n'a aucune importance. Par exemple :

```python
s1 = {"Sabine", "Fred", "Jamy"}
s2 = {"Jamy", "Sabine", "Fred"}
print(s1 == s2) # Affiche "True".
```

— Le mot-clef `in` permet de construire des expressions booléennes dénotant si un élément est présent dans un ensemble. Par exemple :

```python
s = {"Sabine", "Fred", "Jamy"}
print("Fred" in s) # Affiche "True".
print("Frédéric" in s) # Affiche "False".
```

— Python ne contraint pas les éléments d'un même ensemble à être tous du même type. Par exemple, `{"salut", True, (1, 2)}` est une expression valide qui désigne un certain ensemble de longueur 3. Cependant, il est une très bonne pratique d'éviter de manipuler de tels ensembles hétérogènes lorsque cela est possible, c.-à-d. de n'utiliser que des ensembles dont les éléments sont tous d'un même type.

— Un ensemble est un itérable, dont l'ordre d'itération est arbitraire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"Sabine"`, `"Fred"` et `"Jamy"` sur une ligne chacune.

```python
s = {"Sabine", "Fred", "Jamy"}
for x in s:
  print(x)
```

Exercice 157 (Premier exercice sur les ensembles, ⋆)
Soit `s={0, 32, -5, 32, 0, 1}`.

1. *Que vaut* `len(s)` *?*
2. *Que vaut* `(0 in s)` *?*
3. *Que vaut* `("32" in s)` *?*

□

Exercice 158 (Sous-ensemble, ⋆⋆)
Écrire une fonction `subset` *prenant en argument deux ensembles* `s1` *et* `s2`*, et retournant* `True` *ssi* `s1` *est un* sous-ensemble *de* `s2` *(c.-à-d. si tous les éléments de* `s1` *sont des éléments de* `s2`*), et* `False` *sinon. La fonction doit être construite autour d'une boucle for.*

Contrat :

`s1, s2={9, 8, 2}, {1, 2, 9, 7, 8}`	→	*retour :* `True`
`s1, s2={9, 8, 2}, {2, 9, 8}`	→	*retour :* `True`
`s1, s2=set(), {2, 9, 8}`	→	*retour :* `True`
`s1, s2={9, 8, 4}, {1, 2, 9, 7, 8}`	→	*retour :* `False`

□

Création en intension/compréhension [Cours]

— Il est possible de définir un ensemble en utilisant l'expression suivante, dite « en intension » (ou « en compréhension »), à partir d'un itérable de taille finie `it`, d'un nom de variable `x` et d'une expression `exp` : `{exp for x in it}`. Dans l'exemple suivant, l'itérable `it` est `range(6)`, l'expression `exp` est `(2*i)`, et le nom de variable `x` est `i` :

```python
s = {(2*i) for i in range(6)}
print(s) # Affiche "{0, 2, 4, 6, 8, 10}".
```

— La syntaxe précédente crée un ensemble contenant un élément pour chaque élément de l'itérable `it`. Il est possible d'intégrer une condition à une définition en intension pour ne générer des éléments que pour certains éléments de `it`. Par exemple :

```python
s1 = {(2*i) for i in range(6) if((i % 3) == 0)}
print(s1) # Affiche "{0, 6}".

s2 = {(2*i) for i in range(6) if((i % 3) != 0)}
print(s2) # Affiche "{2, 4, 8, 10}".
```

— Une autre manière courante de créer un ensemble est d'appeler la fonction `set` sur un itérable de taille finie. Par exemple :

```python
s1 = set(["Sabine", "Fred", "Jamy"]) # Ensemble à trois éléments.
s2 = set([]) # Ensemble (vide) à zéro élément.
s3 = set(range(10)) # Ensemble à 10 éléments.
s4 = set("abcde") # Ensemble à 5 éléments.
```

Exercice 159 (Longueur d'un ensemble, ⋆)
Soit `it` *un itérable quelconque de taille finie* `n`*, et* `s=set(it)`*.*

1. *Si les éléments de* `it` *sont tous distincts, que vaut* `len(s)` *?*
2. *De manière générale, que peut-on dire de* `len(s)` *?*

□

Exercice 160 (Sont-ce des ensembles ?, ⋆⋆)
Pour chacune des expressions suivantes, dire si elle s'évalue à un ensemble et si oui, en indiquer la longueur et les éléments.

1. `set(range(-2, 10, 3))`
2. `set(["Hello world!"])`
3. `set("Hello world!")`
4. `set([[1, 2], [1, 2]])`
5. `set([(1, 2), (1, 2)])`
6. `set({"Sabine", "Fred", "Jamy"})`
7. `set([{1}, {{1}, 2}])`

□

Exercice 161 (Appartenances, ⋆⋆)

1. *Écrire une fonction* `parthood` *prenant en argument un ensemble* `s` *et une liste* `l`*, et retournant une liste de booléens de même longueur que* `l` *et dont l'élément d'indice* `i` *est* `True` *ssi l'élément d'indice* `i` *de* `l` *est contenu dans* `s`*, et* `False` *sinon. La fonction doit être construite autour d'une boucle for.*

   Contrat :
   `s, l` = `{9, 8, 2}, [0, 2, 2, 10]` → *retour :* `[False, True, True, False]`
2. *Même question, mais en construisant la liste retournée avec une expression en intension.*

□

Opérations fonctionnelles [Cours]

— Si `s1` et `s2` sont deux ensembles, `s1.union(s2)` vaut l'ensemble contenant à la fois les éléments de `s1` et les éléments de `s2` (et uniquement ceux-là). Par exemple :

```python
s = {0, 1, 2, 3}.union({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 1, 2, 3, 4, 6, 8}".
```

— Si `s1` et `s2` sont deux ensembles, `s1.intersection(s2)` vaut l'ensemble contenant les valeurs qui sont à la fois éléments de `s1` et de `s2`. Par exemple :

```
s = {0, 1, 2, 3}.intersection({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 2}".
```

— Si `s1` et `s2` sont deux ensembles, `s1.difference(s2)` vaut l'ensemble contenant les éléments de `s1` qui ne sont pas dans `s2`. Par exemple :

```
s = {0, 1, 2, 3}.difference({0, 2, 4, 6, 8})
print(s) # Affiche "{1, 3}".
```

— Toutes les opérations vues ici sont *fonctionnelles*, c.-à-d. qu'elles génèrent de nouveaux ensembles sans modifier les ensembles initiaux. Par exemple, dans la suite d'instructions suivante, l'ensemble `s1` n'est pas modifié après son initialisation :

```
s1 = {0, 1, 2, 3}
s2 = s1.union({0, 2, 4, 6, 8})
print(s1) # Affiche "{0, 1, 2, 3}".
```

Exercice 162 (Union, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`, *donner la valeur de chacune des expressions suivantes :*

1. `s1.union(s2)`
2. `s1.union(s1)`
3. `s2.union(s1)`
4. `s2.union(s2)`

□

Exercice 163 (Intersection, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`, *donner la valeur de chacune des expressions suivantes :*

1. `s1.intersection(s2)`
2. `s1.intersection(s1)`
3. `s2.intersection(s1)`
4. `s2.intersection(s2)`

□

Exercice 164 (Différence, ⋆)
Soit `s1=set(range(4))` *et* `s2={2, 3, 5, 7}`, *donner la valeur de chacune des expressions suivantes :*

1. `s1.difference(s2)`
2. `s1.difference(s1)`
3. `s2.difference(s1)`
4. `s2.difference(s2)`

□

Opérations non fonctionnelles [Cours]

— Tout comme les listes, les ensembles sont *mutables*, c.-à-d. altérables : il est possible de leur ajouter et de leur supprimer des éléments.

— La méthode `add` permet de rajouter un élément à un ensemble s'il n'y est pas déjà. Par exemple :

```
s = set()

s.add(1)
print(s) # Affiche "{1}".

s.add(2)
print(s) # Affiche "{1, 2}".

s.add(1)
print(s) # Affiche "{1, 2}".
```

— La méthode `remove` permet de supprimer un élément à un ensemble qui le contient. Notons que `remove` lève une exception (≈ le programme plante) si son argument n'est pas un élément de l'ensemble sur lequel elle est appelée.

```
s = {1, 2, 3, 5, 7}

s.remove(1)
print(s) # Affiche "{2, 3, 5, 7}".

s.remove(1) # KeyError
```

— Une autre manière de supprimer un élément à un ensemble est d'utiliser la méthode `discard`. L'intérêt de cette méthode est qu'elle ne lève pas d'exception si son argument n'est pas un élément de l'ensemble sur lequel elle est appelée.

```
s = {1, 2, 3, 5, 7}

s.discard(1)
print(s) # Affiche "{2, 3, 5, 7}".

s.discard(1)
print(s) # Affiche "{2, 3, 5, 7}".
```

— La méthode `update`, appelée sur un ensemble `s1` avec comme argument un ensemble `s2`, rajoute à `s1` les éléments contenus dans `s2`. Par exemple :

```
s = {0, 1, 2, 3}
s.update({0, 2, 4, 6, 8})
print(s) # Affiche "{0, 1, 2, 3, 4, 6, 8}".
```

— L'argument de la méthode `update` peut être n'importe quel itérable de taille finie. Par exemple :

```
s = set()

s.update(range(3))
print(s) # Affiche "{0, 1, 2}".

s.update(range(-3, 5, 2))
print(s) # Affiche "{0, 1, 2, 3, 5, -3, -1}".
```

— Les méthodes `add`, `remove`, `discard` et `update` ne sont pas fonctionnelles. Toutes modifient l'ensemble sur laquelle elles sont appelées et retournent toujours `None`.

Exercice 165 (Caractères numériques, ⋆⋆)
Écrire une fonction `digits` *prenant en argument une chaîne de caractères* `s` *et retournant l'ensemble des caractères numériques (c.-à-d. des chiffres) apparaissant dans* `s`*. Il est possible d'utiliser la méthode* `isdigit` *qui, lorsque appelée (sans argument) sur une chaîne de caractères, retourne* `False` *si cette chaîne est vide ou contient au moins un caractère non numérique (ex : une lettre, une signe de ponctuation), et* `True` *sinon (c.-à-d. si elle n'est pas vide et ne contient que des caractères numériques).*

Contrat :
`s = "Il est 13h42."` → *retour :* `{"1", "2", "3", "4"}`
`s = "Hello world!"` → *retour :* `set()`

□

Exercice 166 (Union et mise-à-jour, ⋆)
Sans utiliser la méthode `union`*, proposer une instruction équivalente à l'instruction suivante.*

```
s1 = s1.union(s2)
```

□

7.2 Les dictionnaires

Le type `dict` [Cours]

— Le type `dict` est le type des structures de données appelées « tableaux associatifs » ou « dictionnaires ». Un dictionnaire est une structure mutable associant une *valeur* à un nombre fini de *clefs* (une valeur par clef).
— Les valeurs d'un dictionnaire peuvent être de n'importe quel type (ex : `int`, `str`, `tuple`, `list`, `set`, `dict`), ce qui n'est pas le cas pour les clefs. Parmi les types natifs, les types non mutables (ex : `int`, `str`, `tuple`) peuvent être utilisés pour les clefs d'un dictionnaire, mais pas les types mutables (ex : `list`, `set`, `dict`).
— Pour accéder à un dictionnaire associant les valeurs `v1`, `v2`, ..., `vn` aux clefs `k1`, `k2`, ..., `kn` respectivement, il est possible d'utiliser l'expression suivante, dite « en extension » : `{k1: v1, k2: v2, ..., kn: vn}`. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
```

— Le dictionnaire vide s'écrit `{}` ou, de manière équivalente, `dict()`.
— Si la même clef apparaît plusieurs fois dans une définition de dictionnaire par extension, seule l'association de sa dernière occurrence sera prise en compte :

```
d = {"Lyon": 0, "Manchester": 1, "Lyon": 2}
print(d == {'Lyon': 2, 'Manchester': 1}) # Affiche "True".
```

— À part cela, l'ordre des couples clef-valeur dans une définition en extension n'est pas pertinent. La raison d'être d'un dictionnaire est simplement d'enregistrer des associations clef-valeur.
— La fonction `print` appelée sur un dictionnaire affiche les associations clef-valeur dans un ordre arbitraire.
— L'on accède à la valeur associée à la clef `k` d'un dictionnaire `d` avec `d[k]`. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d["Edinburgh"]) # Affiche "1".
```

— Si l'on cherche à accéder dans un dictionnaire à la valeur associée à une clef non définie, une erreur (`KeyError`) sera produite à l'exécution.
— La méthode `get`, appelée avec un argument `k`, retourne la valeur associée à la clef `k` si celle-ci est définie et `None` sinon :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d.get("Edinburgh")) # Affiche "1".
print(d.get("Strasbourg")) # Affiche "None".
```

— La méthode `get` accepte optionnellement un second argument, retourné à la place de `None` si le premier argument n'est pas une clef définie :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print(d.get("Strasbourg")) # Affiche "None".
print(d.get("Strasbourg", -1)) # Affiche "-1".
```

— Le nombre d'associations d'un dictionnaire est sa *longueur*. Comme pour une chaîne de caractères ou une liste, on peut accéder à la longueur d'un dictionnaire avec la fonction `len`. Par exemple, si `d={8: "pair", 5: "impair", -4: "pair", 0: "pair"}`, `len(d)` vaut 4.

— Le mot-clef `in` permet de construire des expressions booléennes dénotant si une clef est présente (en tant que clef) dans un dictionnaire. Par exemple :

```
d = {"Lyon": 0, "Manchester": 1, "Firenze": 2, "Marseille": 0,
    "Edinburgh": 1, "Napoli": 2}
print("Edinburgh" in d) # Affiche "True".
print("London" in d) # Affiche "False".
print(2 in d) # Affiche "False".
```

— Python ne contraint ni les clefs ni les valeurs d'un même dictionnaire à être tous du même type. Par exemple, `{"salut": True, 2: "hier", True: [1,2]}` est une expression valide qui désigne un certain dictionnaire de longueur 3. Cependant, il est une très bonne pratique d'éviter de manipuler de tels dictionnaires hétérogènes lorsque cela est possible, c.-à-d. de n'utiliser que des dictionnaires dont les clefs, d'une part, et les valeurs, d'une autre, sont toutes d'un même type.

— Dans un dictionnaire, une seule valeur peut être associée à une clef. Si l'on souhaite intuitivement associer plusieurs valeurs d'un certain type à une même clef, il faut alors utiliser un dictionnaire dont les valeurs sont des n-uplets/listes/ensembles. Par exemple :

```
d = {"France": {"Paris", "Lyon", "Marseille"}, "United Kingdom
    ": {"London", "Birmingham", "Glasgow"}}
```

Exercice 167 (Vérification de clefs, $\star$)
Écrire une fonction `check_keys` *prenant en argument un dictionnaire* `dico` *et une liste* `keys`*, et renvoyant* `True` *si tous les éléments de* `keys` *sont des clefs de* `dico` *et* `False` *sinon.*

Contrat :
`dico, keys` $=$ `{"a": 0, "b": 1, "c": 2}, ["a", "d"]` $\rightarrow$ *retour :* `False`

□

Exercice 168 (Un traducteur, ⋆)
Le but de cet exercice est l'écriture d'un traducteur automatique très primitif. Pour ce traducteur, une phrase est représentée par une liste de tokens*, qui sont des* `str` *représentant des mots ou signes de ponctuation. Ce traducteur traduit une phrase « mot à mot », c.-à-d. en construisant une liste où chaque token de la phrase source est remplacé par son équivalent dans la langue cible d'après un dictionnaire tel que celui-ci :*

```
fr2en = {"langage": "language", "un": "a", "est": "is", "merveilleux
    ": "wonderful"}
```

Quand un token de la phrase source n'est pas trouvé dans le dictionnaire, il n'est pas traduit et est inséré tel quel dans la phrase en sortie. Ce traducteur doit être implémenté sous forme d'une fonction `translate` *prenant en argument un dictionnaire* `src2tgt` *et une phrase* `src_sentence` *(une liste de chaînes de caractères), et retournant le résultat de la traduction.*

> ***Contrat :***
> `src2tgt, src = fr2en, ["Python", "est", "merveilleux", "."]` → *retour :* `["Python", "is", "wonderful", "."]`

□

Exercice 169 (Recherche dans des dictionnaires, ⋆⋆)
On suppose donnés les dictionnaires suivants, l'un associant des noms à des prénoms de personnages (de la série The Big Bang Theory) et l'autre associant des noms d'acteur·rice·s à des noms de personnages :

```
names = {"Leonard": "Hofstadter", "Amy": "Fowler", "Sheldon": "
    Cooper", "Bernadette": "Rostenkow"}
actors = {"Fowler": "Bialik", "Cooper": "Parsons", "Rostenkow": "
    Rauch"}
```

1. *Écrire une fonction* `actor_from_character` *qui prend un prénom* `first_name` *en argument, et qui retourne le nom de l'acteur·rice qui joue ce personnage.*

   > ***Contrat :***
   > `first_name = "Sheldon"` → *retour :* `"Parsons"`

2. *Améliorer la fonction pour qu'elle retourne la chaîne* `"[character name unknown]"` *dans le cas où* `first_name` *n'est pas connu en tant que prénom de personnage, et retourne* `"[actor name unknown]"` *dans le cas où le nom de famille du personnage est connu mais pas le nom de l'acteur·rice qui l'interprète.*

   > ***Contrat :***
   > `first_name = "Penny"` → *retour :* `"[character name unknown]"`
   > `first_name = "Leonard"` → *retour :* `"[actor name unknown]"`
   > `first_name = "Sheldon"` → *retour :* `"Parsons"`

□

Création en intension/compréhension [Cours]

— Il est possible de définir un dictionnaire en utilisant l'expression suivante, dite « en intension » (ou « en compréhension »), à partir d'un itérable de taille finie `it`, d'un nom de variable `x` et de deux expressions `key` et `value` : `{key: value for x in it}`. Dans l'exemple suivant, l'itérable `it` est `range(6)`, l'expression `key` est `i`, l'expression `value` est `(2*i)` et le nom de variable `x` est `i` :

```
d = {i: (2*i) for i in range(6)}
print(d) # Affiche "{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10}".
```

— La syntaxe précédente crée un dictionnaire contenant une association pour chaque élément de l'itérable `it`. Il est possible d'intégrer une condition à une définition en intension pour ne générer des associations que pour certains éléments de `it`. Par exemple :

```python
d1 = {i: (2*i) for i in range(6) if((i % 3) == 0)}
print(d1) # Affiche "{0: 0, 3: 6}".

d2 = {i: (2*i) for i in range(6) if((i % 3) != 0)}
print(d2) # Affiche "{1: 2, 2: 4, 4: 8, 5: 10}".
```

— Une autre manière courante de créer un dictionnaire est d'appeler la fonction `dict` sur un itérable de taille finie de paires. Chaque élément de la paire est alors interprété comme une association clef-valeur, enregistrée dans le dictionnaire ainsi créé. Par exemple :

```python
l = [("Marie", "fr"), ("John", "en"), ("Otto", "de")]
name2language = dict(l)
print(name2language) # Affiche "{'Marie': 'fr', 'John': 'en',
    'Otto': 'de'}".
```

— Pour ces différentes manières de créer un dictionnaire aussi, si la même clef est rencontrée plusieurs fois, seule l'association correspondant à sa dernière occurrence sera conservée.

Exercice 170 (Création d'un dictionnaire par intension, ⋆)

1. *Écrire une fonction* `f` *prenant en argument une liste* `l`*, dont on supposera tous les éléments uniques, et renvoyant le dictionnaire associant à chaque élément de* `l` *sa position dans* `l`*.*
   Contrat :
   `l = ["Sabine", "Fred", "Jamy"]` → *retour :* `{"Sabine": 0, "Fred": 1, "Jamy": 2}`
2. *Que vaut* `f(["Sabine", "Fred", "Sabine", "Jamy"])`*, où* `f` *est la fonction proposée pour la question précédente ?*
3. *Plus généralement, si l'on ne se restreint pas à des listes dont tous les éléments sont uniques, dans* `f(l)`*, quelle est la valeur associée à chaque élément de* `l` *?*

□

7.3 Opérations sur les dictionnaires

Parcours de dictionnaires [Cours]

— La méthode `keys` permet d'accéder à un itérable contenant les clefs d'un dictionnaire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"Paris"`, `"London"` et `"Berlin"` sur une ligne chacune.

```python
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d.keys():
  print(k)
```

— La méthode `values` permet d'accéder à un itérable contenant les valeurs d'un dictionnaire. Par exemple, l'exécution de la suite d'instructions suivantes affichera, dans un ordre arbitraire, les chaînes `"France"`, `"United Kingdom"` et `"Deutschland"` sur une ligne chacune.

```python
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for v in d.values():
  print(v)
```

— En Python, un dictionnaire est un itérable dont les éléments sont ses clefs. Par exemple, la suite d'instructions suivante est équivalente à celle mentionnée plus haut en introduction de la méthode `keys`.

```python
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d:
  print(k)
```

— Pour éviter de faire des erreurs et faciliter la lecture du code, je vous recommande de ne jamais itérer directement sur un dictionnaire mais d'utiliser plutôt la méthode `keys`. Je vous recommande aussi de choisir des noms de variables appropriés pour les compteurs de vos boucles, comme dans les exemples précédents.

— La méthode `items` permet d'accéder à un itérable contenant les paires clefs-valeurs d'un dictionnaire. Ainsi, les deux suites d'instructions suivantes sont équivalentes :

```python
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for (k, v) in d.items()
  print(f"La ville nommée '{k}' est la capitale du pays nommé
    '{v}'.")
```

```python
d = {"Paris": "France", "London": "United Kingdom", "Berlin":
    "Deutschland"}
for k in d.keys()
  print(f"La ville nommée '{k}' est la capitale du pays nommé
    '{d[k]}'.")
```

Exercice 171 (Clefs disjointes, ⋆⋆)
Écrire une fonction `disjoint_keys` *prenant en argument deux dictionnaires* `d1` *et* `d2`*, et retournant* `True` *ssi les clefs des* `d1` *et* `d2` *sont disjointes (c.-à-d. si les deux dictionnaires n'ont aucune clef en commun), et* `False` *sinon.*

Contrat :
`d1, d2 = {"Li": 3, "H": 1, "He": 2}, {"Ne": 20.2, "F": 9.0, "O": 16.0}` → *retour :* `True`
`d1, d2 = {"H": 1, "He": 2}, {"Ne": 20.2, "He": 4.0, "Ar": 40.0}` → *retour :* `False`

□

Exercice 172 (Inversion d'un dictionnaire, ⋆⋆)
Écrire une fonction `inverse` *prenant en argument un dictionnaire* `d` *et retournant le dictionnaire dont les associations clef-valeur sont les inverses des associations clef-valeur de* `d`.

Contrat :
`d = {"Sabine": "Quindou", "Frédéric": "Courant", "Jamy": "Gourmaud"}` → *retour :* `{"Quindou": "Sabine", "Courant": "Frédéric", "Gourmaud": "Jamy"}`

□

Exercice 173 (Du bon usage des dictionnaires, ⋆⋆)

1. *Quel est l'affichage produit par la suite d'instructions suivante ?*

```python
counts = {"tree": 12, "river": 7, "house": 9}
form_tgt = "river"
for form, count in counts.items():
  if(form == form_tgt):
    print(f"Count for '{form}': {count}")
```

2. *Simplifier la suite d'instructions précédente sans utiliser de boucle.*
3. *Simplifier la suite d'instructions suivante.*

```python
counts = {"tree": 12, "river": 7, "house": 9}
form_tgts = ["tree", "river"]
for form_tgt in form_tgts:
  for form, count in counts.items():
    if(form == form_tgt):
      print(f"Count for '{form_tgt}': {count}")
```

Moralité : Il ne faut jamais itérer sur un dictionnaire si l'on cherche une clef particulière. Les exemples donnés ici à simplifier sont la marque d'une incompréhension profonde de l'usage des dictionnaires.

□

Modification de dictionnaires [Cours]

— Tout comme les listes ou les ensembles, les dictionnaires sont *mutables*, c.-à-d. altérables : leurs associations et le nombre de ces associations sont modifiables.

— Soit un dictionnaire `d`, une clef `k` définie ou non dans `d` et `v` une valeur, l'instruction d'assignation `d[k] = v` associe la valeur `v` à la clef `k`, écrasant le cas échéant l'association précédente. Par exemple :

```python
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

d["Italia"] = "Roma"
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Rio de Janeiro', 'Italia': 'Roma'}".

d["Brasil"] = "Brasilia" # En 1960.
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Brasilia', 'Italia': Roma}".
```

La méthode `update`, appelée sur un dictionnaire `d1` avec comme argument un dictionnaire `d2`, rajoute à `d1` les associations contenues dans `d2`, écrasant les éventuelles associations conflictuelles initiales. Par exemple :

```python
d1 = {"France": "Paris", "United Kingdom": "London", "Brasil":
     "Rio de Janeiro"}
d2 = {"Italia": "Roma", "Brasil": "Brasilia"}

d1.update(d2)
print(d1) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London', 'Brasil': 'Brasilia', 'Italia': 'Roma'}".
```

— Il est possible d'effacer une association de clef `k` dans un dictionnaire `d` avec l'instruction `del d[k]`. Notons que l'exécution de cette instruction lève une exception si `k` n'est pas une clef définie dans `d`. Par exemple :

```python
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

del d["Brasil"]
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".

del d["Brasil"] # KeyError
```

— Une autre manière d'effacer une association de clef `k` dans un dictionnaire `d` est d'utiliser l'instruction `d.pop(k)`. Si `k` est effectiment une clef de `d`, l'expression `d.pop(k)` est évaluée à la valeur qui lui

est associée avant l'effacement de l'association ; sinon, une exception (`KeyError`) est levée.

— L'un des avantages de la méthode pop tient au fait qu'elle accepte optionnellement un second argument : l'instruction `d.pop(k, None)` efface dans `d` l'association de clef `k` et retourne la valeur correspondante si elle existe, et retourne `None` (sans erreur) sinon. Par exemple :

```
d = {"France": "Paris", "United Kingdom": "London", "Brasil":
    "Rio de Janeiro"}

print(d.pop("Brasil", None)) # Affiche "Rio de Janeiro".
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".

print(d.pop("Brasil", None)) # Affiche "None".
print(d) # Affiche "{'France': 'Paris', 'United Kingdom': '
    London'}".
```

Exercice 174 (Définir un dictionaire, ⋆⋆)
Écrire une fonction `associate` *prenant en argument deux listes* `keys` *et* `values`*, et retournant le dictionnaire associant chaque élément de* `values` *à l'élément de même indice dans* `keys` *si (i) ces deux listes sont de même longueur et (ii) les éléments de* `keys` *sont tous distincts, et* `None` *sinon.*

Contrat :
`keys, values = ["e", "a", "z", "d"], [5, 1, 26, 4]` → *retour :* `{"e": 5, "a": 1, "z": 26, "d": 4}`
`keys, values = ["e", "a", "z"], [5, 1, 26, 4]` → *retour :* `None`
`keys, values = ["e", "a", "e", "z"], [5, 1, 26, 4]` → *retour :* `None`

□

Exercice 175 (Compter les occurrences, ⋆⋆)
Écrire une fonction `count` *prenant en argument une liste* `l` *et retournant un dictionnaire associant à chaque élément de* `l` *son nombre d'occurrences dans* `l`*.*

Contrat :
`l = ["h", "e", "l", "l", "o"]` → *retour :* `{"h": 1, "e": 1, "l": 2, "o": 1}`

□

Exercice 176 (Ensemble des occurrences, ⋆⋆)
Écrire une fonction `positions` *prenant en argument une liste* `l` *et retournant un dictionnaire associant à chaque élément de* `l` *l'ensemble des indices de ses occurrences dans* `l`*.*

Contrat :
`l = ["h", "e", "l", "l", "o"]` → *retour :* `{"h": {0}, "e": {1}, "l": {2, 3}, "o": {4}}`

□

7.4 Exercices supplémentaires

Exercice 177 (Faux amis, ⋆⋆)
On dit de deux mots de deux langues différentes qu'ils sont des faux amis *si ces deux mots ont des sens clairement distincts mais s'orthographient de manière tellement similaire que l'on pourrait croire (à tort) qu'il s'agit de traductions l'un de l'autre. C'est le cas, par exemple, de « actually » en anglais et « actuellement » en français ; « actually » se traduit généralement par « en fait » et « actuellement » par « currently ». Un autre exemple est le cas de « eventually » en anglais et « éventuellement » en français, qui se traduisent généralement par « finalement » et « possibly », respectivement.*

Dans cet exercice, on va s'intéresser aux faux amis exacts, *qui sont des mots ayant exactement la même orthographe, comme « coin »/« coin » ou « figure »/« figure ».*

Écrire une fonction `faux_amis` *prenant en argument un dictionnaire* `src2tgt` *dont les paires clef-valeurs consistent en un mot dans une langue source et sa traduction dans une langue cible, et retournant l'ensemble de tous les* faux amis exacts *qui existent dans ce dictionnaire.*

> ***Contrat :***
> `src2tgt` = `{"avion": "plane", "coin": "corner", "pièce": "coin", "zoo": "zoo"}` → *retour :* `{"coin"}`

□

Exercice 178 (Composition de dictionnaires, ⋆⋆)
Écrire une fonction `compose_dict` *prenant en argument deux dictionnaires* `d1` *et* `d2`*, et retournant le dictionnaire contenant les associations clef-valeurs* `k: v` *telles qu'il existe* `x` *tel que* `k: x` *est une association de* `d1` *et* `x: v` *est une association de* `d2`*.*

> ***Contrat :***
> *Soit les trois dictionnaires suivants,*

```
fr2en = {"maison": "house", "rue": "road", "lac": "lake"}
en2de = {"house": "Haus", "road": "Strasse", "tower": "Turm"}
fr2de = {"maison": "Haus", "rue": "Strasse"}
```

> `d1, d2` = `fr2en, en2de` → *retour :* `fr2de`

□

Exercice 179 (Associations clef-clef', ⋆⋆⋆)
Imaginons que l'on souhaite non plus enregistrer de simples associations clef-valeur comme dans un dictionnaire, où chaque clef est associée à exactement une valeur et où plusieurs clefs peuvent être associées à la même valeur, mais des associations « clef-clef' », dans le sens où si la clef `k_l` *est associée à la clef'* `k_r`, `k_l` *n'est associée à aucune autre clef' que* `k_r` *et aucune autre clef que* `k_l` *n'est associée à la clef'* `k_r`.

Pour représenter une telle structure de donnée, nous allons utiliser deux dictionnaires `d1` *et* `d2` *tels qu'une association clef-clef'* `k_l: k_r` *soit représentée par une association* `k_l: k_r` *dans* `d1` *ainsi qu'une association* `k_r: k_l` *dans* `d2`.

Écrire une fonction `change` *prenant en argument deux tels dictionnaires* `d1` *et* `d2`, *une clef* `k_l` *et une clef'* `k_r`, *et modifiant* `d1` *et* `d2` *de manière à enregistrer l'association* `k_l: k_r` *en écrasant le cas échéant toute association conflictuelle.*

Contrat :
`d1, d2, k_l, k_r = {}, {}, "a", 1 → d1, d2 = {"a": 1}, {1: "a"}`
`d1, d2, k_l, k_r = {"a": 1}, {1: "a"}, "b", 2 → d1, d2 = {"a": 1, "b": 2}, {1: "a", 2: "b"}`
`d1, d2, k_l, k_r = {"a": 1, "b": 2}, {1: "a", 2: "b"}, "c", 2 → d1, d2 = {"a": 1, "c": 2}, {1: "a", 2: "c"}`
`d1, d2, k_l, k_r = {"a": 1, "c": 2}, {1: "a", 2: "c"}, "a", 2 → d1, d2 = {"a": 2}, {2: "a"}`

□

Chapitre 8

Manipulation de fichiers

Objectifs :

- Lister le contenu d'un dossier.
- Tester l'existence d'un fichier ou dossier.
- Lire un fichier texte.
- Savoir détecter la fin d'un fichier texte.
- Écrire dans un fichier texte.

8.1 Le système de fichiers

Le système de fichiers [Cours]

- Les données enregistrées sur un support physique (ex : un disque dur, une clef usb) sont généralement représentées sous forme d'une structure constituée de *dossiers* (ou « répertoires ») et de *fichiers*.
- Un fichier peut contenir tout type de données alors que les dossiers ne servent a priori qu'à l'organisation des fichiers. Un dossier peut être vu comme une liste (éventuellement vide) de fichiers et de dossiers formant son *contenu*. Excepté un dossier particulier, la *racine du système de fichiers*, tout dossier et tout fichier est *contenu* dans un unique dossier, son *dossier parent*.
- Les *descendants* d'un dossier sont (par récurrence) :
  - tout fichier ou dossier qu'il contient ;
  - tout descendant d'un dossier qu'il contient ;
- La structure obtenue est *arborée* :
  - elle ne contient pas de boucle, dans le sens où un dossier n'est jamais l'un de ses propres descendants ;
  - il existe un dossier (la racine), dont tous les autres éléments descendent.
- Tout dossier et tout fichier possède un *nom*, représenté en Python par une chaîne de caractères (ex : `"documents"`, `"exam.pdf"`). Le nom de la racine correspond généralement à la chaîne vide et tous les autres éléments du système de fichiers ont des noms non vides. Les éléments contenus dans un même dossier doivent avoir des noms tous distincts.
- Il est courant pour un processus interagissant avec le système de fichiers (ex : un terminal, un navigateur de fichiers) de donner un statut particulier à un dossier, alors appelé « répertoire de travail actuel ». Lorsque Python exécute du code, un répertoire de travail actuel est défini. Le choix initial de ce dossier dépend de la manière dont le code a été exécuté.

Chemins [Cours]

- Un *chemin* est une chaîne de caractères non vide désignant (ou « pointant vers ») un élément existant ou fictif dans un système de fichiers. Un chemin désignant un fichier existant peut par exemple servir à lire ce fichier. Un chemin désignant un élement fictif peut par exemple servir à créer un élement à cet emplacement.

— Pour interpréter un chemin, il est nécessaire de définir trois valeurs de type `str` (qui peuvent varier d'un système à un autre) :
  — le *séparateur* (`"/"` dans les exemples qui suivent) ;
  — l'*abréviation de répertoire courant* (`"."` dans les exemples qui suivent) ;
  — l'*abréviation de répertoire parent* (`".."` dans les exemples qui suivent).
  Pour interpréter certains chemins, dit « relatifs », il est aussi nécessaire d'avoir une notion de répertoire de référence. C'est généralement le répertoire de travail actuel qui va servir de référence pour l'interprétation des chemins relatifs en PYTHON.
— Quelques exemples :
  — le chemin `"/home/guest/documents"` désigne un élément `"documents"` situé dans le dossier `"guest"` situé dans le dossier `"home"` de la racine ;
  — le chemin `"./../../exam.pdf"` désigne l'élément `"exam.pdf"` situé dans le dossier parent du dossier parent du répertoire de référence ;
  — le chemin `"../../exam.pdf"` désigne exactement le même élément que le chemin précédent ;
  — le chemin `"/"` désigne la racine du système de fichiers.
— Formellement, pour interpréter un chemin, il faut le décomposer en fonction des occurrences du séparateur. Nous reviendrons plus bas sur la méthode `split`, mais `path.split(sep)` est une liste de chaînes de caractères : la liste des chaînes obtenues en découpant `path` au niveau de chaque occurrence de `sep`. Par exemple, `"./../../exam.pdf".split("/")` vaut `[".", "..", "..", "exam.pdf"]`, `"/home/guest/documents".split("/")` vaut `["", "home", "guest", "documents"]` et `"/".split("/")` vaut `["", ""]`.
— On peut définir l'élément désigné par un chemin `path` étant donné le séparateur `sep` par récurrence sur la liste `path.split(sep)` :
  — cas de base :
    — `[""]` désigne la racine,
    — `["."]` désigne le répertoire de référence,
    — `[".."]` désigne le parent du répertoire de référence ou le répertoire de référence lui-même s'il s'agit de la racine,
    — pour toute autre chaîne de caractères `s` ne contenant pas `sep`, `[s]` désigne l'élément de nom `s` contenu dans le répertoire de référence ;
  — si la liste `l` désigne un dossier `d`, alors :
    — `l + [""]` désigne encore `d`,
    — `l + ["."]` désigne encore `d`,
    — `l + [".."]` désigne le parent de `d` ou `d` s'il s'agit de la racine,
    — pour tout autre chaîne de caractères `s` ne contenant pas `sep`, `l + [s]` désigne l'élément de nom `s` contenu dans `d`.
— Les chemins commençant directement par le séparateur (ex : `"/home/guest/documents"`) sont *absolus* : leur interprétation ne dépend pas du répertoire de référence. Les autres chemins sont les chemins relatifs (ex : `"../../exam.pdf"`), dont l'interprétation dépend du répertoire de référence.

8.2 Lecture du système de fichiers

Lecture de l'arborescence de fichiers [COURS]

— Un grand nombre de fonctions et autres valeurs utiles pour la manipulation du système de fichiers sont disponibles en PYTHON dans le module (≈ la bibliothèque) `os`. On peut accéder à ces fonctions et valeurs après avoir exécuté l'instruction suivante :

```
1 import os
```

— Il est possible de connaître à tout moment le chemin absolu du répertoire de travail actuel (sous forme d'une chaîne de caractères) en évaluant l'expression `os.getcwd()` :

```
1 print(os.getcwd()) # Affiche "/home/guest".
```

— Le nom de la fonction `getcwd` est une abréviation de « *get current working directory* ».

— La valeur du séparateur est assignée à la variable `os.sep`. La valeur de l'abréviation de répertoire courant est assignée à la variable `os.curdir`. La valeur de l'abréviation de répertoire parent est assignée à la variable `os.pardir`.

```
print(os.curdir) # Affiche ".".
print(os.pardir) # Affiche "..".
print(os.sep) # Affiche "/".
```

— Comme la valeur des trois chaînes précédentes peut varier d'un système à l'autre, afin que votre code soit portable (c.-à-d. puisse s'exécuter facilement sur différentes machines), je vous demande de toujours passer par ces trois noms de variable et non vers leurs valeurs directement (ce qui serait un exemple de « codage en dur »).

— En pratique, `os.sep` est surtout utilisée comme chemin absolu vers la racine. Pour construire des chemins à partir d'autres chemins ou noms de fichier/dossier, il est possible d'utiliser la fonction `os.path.join`. Par exemple :

```
path = os.sep # Désigne la racine.
print(path) # Affiche "/".
path = os.path.join(path, "home", "guest")
print(path) # Affiche "/home/guest".
```

```
path = os.curdir # Désigne le répertoire de travail actuel.
print(path) # Affiche ".".
path = os.path.join(path, "documents", os.pardir, "videos", "
    films")
print(path) # Affiche "./documents/../videos/films".
```

— Il est possible de lister le contenu d'un dossier à l'aide de la fonction `os.listdir`. Si `path` est un chemin pointant vers un dossier `d`, alors `os.listdir(path)` vaut la liste des noms des éléments contenus dans `d`. Par exemple, les instructions suivantes, qui sont équivalentes, affichent le contenu du répertoire de travail actuel :

```
for e_name in os.listdir(os.getcwd()): print(e_name)
```

```
for e_name in os.listdir(os.curdir): print(e_name)
```

Notons que `os.listdir` lève une exception ($\approx$ le programme plante) si son argument pointe vers un fichier (plutôt qu'un dossier) ou vers un élément fictif.

— La fonction `os.path.exists` permet de savoir si son argument est un chemin pointant vers un élément existant. La fonction `os.path.isfile`, elle, ne vaut `True` que si le chemin pointe vers un fichier, alors que la fonction `os.path.isdir` ne vaut `True` que si le chemin pointe vers un dossier. Par exemple, si le système de fichiers contient à sa racine un dossier `"home"`, contenant un dossier `"guest"`, contenant un fichier `"chap_6.pdf"` :

```
path = os.path.join(os.sep, "home", "guest", "chap_6.pdf")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "True".
print(os.path.isdir(path)) # Affiche "False".

path = os.path.join(os.sep, "home", "guest")
print(os.path.exists(path)) # Affiche "True".
print(os.path.isfile(path)) # Affiche "False".
print(os.path.isdir(path)) # Affiche "True".
```

Exercice 180 (Afficher les dossiers à la racine, $\star$)
Écrire une suite d'instructions affichant les noms des dossiers (et non des fichiers) contenus à la racine du

système de fichiers. □

Exercice 181 (Afficher des fichiers, ⋆)
Écrire une fonction `printdir` *prenant en argument un chemin supposé pointant vers un dossier (existant) et ne retournant rien mais affichant les noms des fichiers (et non des dossiers) contenus dans ce dossier.* □

Lecture de fichiers [Cours]

— La fonction `open` permet d'ouvrir un fichier afin d'y lire ou écrire du contenu. Cette fonction renvoie une *interface d'entrée/sortie* (« IO wrapper ») possédant des méthodes de lecture/écriture de contenu. Pour éviter certains effets indésirables, il faut *fermer* toute interface ouverte dès qu'elle n'est plus utile, avec la méthode `close`.

— Tout fichier texte utilise un format d'encodage (c.-à-d. une manière de convertir chaque symbole en une séquence de bits). Dans ce cours, nous supposons que tous les textes sont encodés en `UTF-8` — un format d'encodage capable d'encoder la totalité des caractères définis par le standard Unicode (plus de 110.000) et utilisé aujourd'hui par la quasi-totalité des pages web.

— Pour lire un fichier texte, il est possible de l'ouvrir en fournissant à `open` trois arguments :

1. un chemin pointant vers ce fichier ;
2. la chaîne `"r"` (« read »),
3. la chaîne `"utf-8"`, pour l'argument spécial `encoding`.

Le deuxième argument de `open` est appelé « mode d'ouverture ».

— Notons qu'avec le mode d'ouverture `"r"`, `open` lève une exception (≈ le programme plante) si son premier argument n'est pas un chemin vers un fichier texte existant.

— Python associe à toute interface d'entrée/sortie une position dans le fichier appelée « position courante ». Lorsque l'on lit ou écrit dans un fichier via une interface, on y lit ou écrit à la position courante. Lorsqu'un fichier est ouvert en mode `"r"`, la position courante est initialisée au tout début du fichier.

— La méthode `read` appelée sur l'interface ainsi obtenue renvoie, sous forme d'une chaîne de caractères, tout le contenu du fichier *à partir de la position courante*. La position courante est aussi déplacée par `read`, à la fin du fichier. Par exemple, si `filename` représente un chemin pointant vers un fichier texte constitué des deux lignes de texte `"Bonjour.\nHello.\n"` :

```
f = open(filename, "r", encoding="utf-8") # Ouverture pour
    lecture. Position courante initialisée au début du fichier.

text = f.read() # "Bonjour.\nHello\n" ; la position courante
    est déplacée à la fin du fichier.
print(text, end="")

f.close() # Fermeture.
```

— La position courante étant déplacée par `read` à la fin du fichier, si l'on appelle deux fois de suite cette méthode sur la même interface, le deuxième appel retourne nécessairement la chaîne vide (`""` ; il s'agit bien du contenu du fichier se situant entre la position du courante, se trouvant alors être la fin du fichier, et la fin du fichier).

— Plutôt que d'avoir à explicitement fermer l'interface d'entrée/sortie avec la méthode `close`, il est possible d'ouvrir le fichier avec une construction en `with`. Par exemple, les deux blocs de code suivants sont équivalents :

```
f = open(filename, "r", encoding="utf-8")

print("[début de la lecture]")
print(f.read(), end="")
print("[fin de la lecture]")

f.close() # Fermeture.
```

```
with open(filename, "r", encoding="utf-8") as f: # f est
    automatiquement fermé après l'exécution du bloc de code
    introduit.
  print("[début de la lecture]")
  print(f.read(), end="")
  print("[fin de la lecture]")
```

— La méthode `readline` permet de lire un fichier texte ligne par ligne. `readline` déplace la position courante jusqu'après le prochain caractère de fin de ligne `"\n"` ou la fin du fichier s'il n'y en a plus, et retourne tout le texte se situant entre les deux. Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r", encoding="utf-8") as f:
  print(f.readline(), end="") # Affiche la première ligne du
    fichier.
  print(f.readline(), end="") # Affiche la seconde ligne du
    fichier.
  print(f.readline(), end="") # Affiche la troisième ligne du
    fichier.
  print(f.readline(), end="") # Aucun effet.
  print(f.readline(), end="") # Aucun effet.
```

— La méthode `readlines` fonctionne de manière similaire à `read`, dans le sens où elle lit le fichier de la position courante jusqu'à la fin et y déplace la position courante. Cependant, `readlines` ne retourne pas le contenu lu sous forme d'une unique chaîne de caractères, mais d'une liste de lignes de texte (incluant les caractères de fin de ligne). Par exemple, si `filename` représente un chemin pointant vers un fichier constitué d'exactement trois lignes de texte :

```
with open(filename, "r", encoding="utf-8") as f:
  l1 = f.readlines() # Liste de trois chaînes de caractères.
  print(len(l1)) # Affiche "3".
  l2 = f.readlines() # Liste vide.
  print(len(l2)) # Affiche "0".
```

Exercice 182 (Type des valeurs de retour, ⋆)

1. *Quel est le type de la valeur d'un appel à la méthode* `read` *?*
2. *Quel est le type de la valeur d'un appel à la méthode* `readline` *?*
3. *Quel est le type de la valeur d'un appel à la méthode* `readlines` *?*

□

Exercice 183 (Affichage de certaines lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et affichant (sans saut de ligne supplémentaire dû à la fonction* `print`*) chacune des lignes ne commençant pas par* `"#"`*.*
2. *Modifier la fonction afin que rien ne se passe (en particulier, pas d'erreur) si l'argument* `filename` *passé ne pointe pas vers un fichier existant.*

□

Exercice 184 (Réimplémentation de readlines, ⋆⋆)
Écrire une fonction `readlines` *réimplémentant la méthode du même nom, c.-à-d. prenant en argument* `f`*, l'interface d'entrée/sortie obtenue à l'ouverture d'un fichier texte en lecture, et retournant la liste de toutes les lignes qui restent à y être lues. La fonction ne doit pas utiliser la méthode* `readlines` *mais plutôt* `readline`*.* □

Chargement en mémoire ; ligne de texte [Cours]

— Il peut être plus pratique d'utiliser `readlines` que `readline`. Cependant, alors qu'un appel à `readline` ne provoque la lecture et mise en mémoire que d'au plus une ligne de texte, un appel à `readlines` (comme à `read`) provoque la lecture et mise en mémoire de tout le contenu du fichier à partir de la position courante. Dans certaines situations, typiquement parce que ce contenu est trop volumineux ou parce que seule une ligne est utile, il est préférable voire nécessaire d'utiliser `readline`.

— Noter qu'un fichier texte non vide, pour être *correctement formé*, doit se terminer par un caractère de fin de ligne `"\n"`. Il s'agit là d'une convention qui n'est malheureusement pas toujours vérifiée mais que je vous demande de respecter. Par exemple, nous supperons qu'un fichier texte ne peut pas avoir pour contenu `"Bonjour.\nHello."`, car celui-ci ne se termine pas par un caractère de fin de ligne.

— D'après cette même convention, une *ligne de texte* est par définition une chaîne de caractères se terminant par un caractère de fin de ligne, et ne contenant aucun autre caractère de fin de ligne.

— Au passage, la valeur d'un appel à la fonction `input` est donc une ligne de texte (provenant généralement non pas d'un fichier texte mais d'une saisie clavier).

— Une ligne de texte vide est matérialisée par la chaîne `"\n"`. Cette chaîne est donc différente de la chaîne vide (`""`), retournée par `read` et `readline` lorsque la position courante est déjà à la fin du fichier.

— Rappelons que la position courante associée à une interface d'entrée/sortie est unique, dans le sens où il n'y a pas de positions courantes différentes utilisées par des méthodes (ex : `read`, `readline`) différentes.

Exercice 185 (Fichier vide ?, ⋆)
Écrire une fonction `isEmpty` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin pointant vers un fichier texte (existant), et retournant* `True` *si ce fichier est vide,* `False` *sinon. Écrire une fonction ne nécessitant pas forcément la lecture de tout le fichier texte.* □

Exercice 186 (Manipulation des méthodes de base, ⋆)
Considérer que `"/home/guest/documents/dialog.txt"` *pointe vers un fichier texte contenant exactement les deux lignes suivantes (incluant les tirets) :*

```
- Bonjour, comment allez-vous ?
- Très bien, et vous ?
```

Décrire très précisément (c.-à-d. notamment sans oublier les éventuelles lignes vides) l'affichage produit par l'exécution de chacune des suites d'instructions suivantes.

1.
```python
import os
path = os.path.join(os.sep, "home", "guest", "documents", "
    dialog.txt")
with open(path, "r", encoding="utf-8") as f:
  print(f.read())
  print(f.readline())
  print(f.readlines())
```

2.
```python
import os
path = os.path.join(os.sep, "home", "guest", "documents", "
    dialog.txt")
with open(path, "r", encoding="utf-8") as f:
  print(f.readline())
  print(f.readlines())
  print(f.read())
```

□

Fonctions utiles au traitement de fichiers texte [Cours]

— La méthode `lstrip` (« left strip ») appelée sans argument sur une chaîne de caractères `s` retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement (ex : espace, tabulation) et de saut de ligne situés *à gauche* du premier caractère autre (c.-à-d. qui n'est ni un caractère d'espacement ni un saut de ligne). La méthode `rstrip` (« right strip ») a un comportement similaire mais supprime les caractères situés *à droite* du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.lstrip() # "Bonjour. Comment-allez vous ?   \t    \n"
s3 = s1.rstrip() # " \t\t  \n   Bonjour. Comment-allez vous ?"
```

— La méthode `strip` peut être vue comme une combinaison de `rstrip` et de `lstrip`. Appelée sans argument sur une chaîne de caractères `s`, elle retourne la chaîne de caractères obtenue en supprimant de `s` tous les caractères d'espacement et de saut de ligne situés à gauche du premier caractère autre et tous ceux situés à droite du dernier caractère autre. Par exemple :

```
s1 = " \t\t  \n   Bonjour. Comment-allez vous ?   \t    \n"
s2 = s1.strip() # "Bonjour. Comment-allez vous ?"
```

— Les méthodes `strip`/`rstrip` sont particulièrement utiles pour éliminer le caractère `"\n"` situé à la fin d'une ligne lue depuis un fichier texte.

— Par défault, ces trois méthodes suppriment les caractères d'espacement et de saut de ligne, mais il est possible de spécifier l'ensemble des caractères à supprimer en leur passant comme argument une chaîne de caractères composée des caractères à supprimer. Par exemple :

```
s1 = "aabbabaccabcdddbaaab"
s2 = s1.strip("ab") # "ccabcddd"
s3 = s1.strip("abcd") # ""
```

— La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau de chaque occurrence de `sep`. Par exemple :

```
s = "bloup--blip-bloup--blop"
l = s.split("--") # ["bloup", "blip-bloup", "blop"]
```

```
s = "bloup--blip-bloup--blop"
l = s.split("-") # ["bloup", "", "blip", "bloup", "", "blop"]
```

— Il faut garder en tête que l'argument de `split` est interprété de manière très différente de l'argument de `strip` et ses variantes :
  — `s.split(sep)` découpe `s` au niveau de chaque occurrence (complète) de `sep`.
  — `s.strip(chars)` retourne une chaîne obtenue en supprimant de `s` des occurrences, non pas de `chars`, mais de n'importe quel caractère présent dans `chars`.

— La méthode `split` est la réciproque de `join`, dans le sens où si `s` et `sep` sont deux chaînes de caractères, `sep.join(s.split(sep))` vaut `s`.

— La méthode `split` est utile pour séparer (approximativement) les différents *tokens* (c.-à-d. occurrences de mots) d'une phrase en français. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ") # ["Le", "chat", "court", "après", "la", "
    souris."]
```

— La méthode `split` accepte un argument supplémentaire indiquant un nombre maximum de coupes à effectuer. La méthode `split` appelée sur une chaîne de caractères `s` avec pour argument une chaîne de caractères `sep` et un entier `n` retourne la liste de chaînes de caractères obtenue en découpant `s` au niveau des **`n` premières occurrences de `sep`**. Par exemple :

```
s = "Le chat court après la souris."
l = s.split(" ", 3) # ["Le", "chat", "court", "après la souris."]
l = s.split(" ", 0) # ["Le chat court après la souris."]
```

Exercice 187 (Découpage de noms, ⋆⋆)

1. *Écrire une fonction* `split_name` *prenant en argument une chaîne de caractères* `name` *supposée contenant un unique espace, et retournant la paire (un* `tuple`*) composée des deux sous-chaînes de* `name` *obtenues en la découpant au niveau de cet espace.*

   Contrat :
   `name = "George Sand"` → *retour :* `("George", "Sand")`

2. *Modifier la fonction proposée à la question précédente pour qu'elle renvoie* `None` *si* `name` *ne contient pas un unique espace.*

   Contrat :
   `name = "George Sand Michael"` → *retour :* `None`
   `name = "George"` → *retour :* `None`
   `name = "George Sand"` → *retour :* `("George", "Sand")`

3. *Modifier la fonction proposée à la question précédente pour qu'elle ne renvoie plus une paire* `(s1, s2)` *lorsque* `name` *contient un unique espace mais un dictionnaire de clefs* `"prénom"` *et* `"nom"`*,* `{"prénom": s1, "nom": s2}`*.*

   Contrat :
   `name = "George Sand"` → *retour :* `{"prénom": "George", "nom": "Sand"}`

□

Exercice 188 (Création de vocabulaire, ⋆⋆)
Écrire une fonction `words_set` *prenant en argument une chaîne de caractères* `filename` *supposée représenter le chemin d'un fichier texte, et retournant l'ensemble des mots apparaissant dans ce fichier. (Supposer que tous les tokens sont séparés par des espaces.)* □

8.3 Écriture du système de fichiers

Écriture de fichiers [Cours]

— Si l'on appelle la fonction `open` avec comme arguments
  — un chemin pointant vers un fichier fictif mais situé dans un dossier existant
  — le mode `"w"` (« write »),
  — et l'encodage `"utf-8"`,
  alors un fichier texte encodé en UTF-8 est créé à l'emplacement pointé par le chemin (avec un contenu vide) et une interface d'entrée/sortie permettant d'y écrire du contenu est retournée. Par exemple, si le répertoire de travail actuel contient un dossier `"documents"` qui lui ne contient pas de fichier `"test.txt"`, alors l'exécution des instructions suivantes crée un tel fichier, entièrement vide :

```
path = os.path.join(os.curdir, "documents", "test.txt")
with open(path, "w", encoding="utf-8") as f: # Ouverture pour écriture.
  print("Empty file created.")
```

— Si le chemin passé comme premier argument à `open` avec le mode `"w"` pointe vers un fichier existant, alors ce fichier est *écrasé*, c.-à-d. que son contenu est effacé.
— Dans tous les cas, l'interface d'entrée/sortie retournée par `open` en mode `"w"` correspond donc à un fichier vide ; la position courante est définie au début du fichier, qui se trouve être aussi la fin.

— La méthode `write`, appelée sur une interface ouverte en écriture et avec pour argument une chaîne de caractères `s`, écrit `s` à partir de la position courante et déplace celle-ci jusqu'après le dernier caractère inséré. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` contient exactement la ligne de texte `"Hello world!\n"` :

```
with open(filename, "w", encoding="utf-8") as f:
  f.write("Hell")
  f.write("o world!")
  f.write("\n")
```

— Contrairement à `print` qui, par défaut, affiche à l'écran un saut de ligne supplémentaire après son argument, la méthode `write` n'écrit dans un fichier que les sauts de ligne explicitement contenus dans son argument. Par exemple, après exécution des instructions suivantes, le fichier pointé par `filename` n'est pas un fichier texte correctement formé car son contenu se termine par un caractère qui n'est pas un caractère de saut de ligne :

```
with open(filename, "w", encoding="utf-8") as f:
  f.write("Hello world!")
```

— Il est possible d'utiliser `open` avec le mode `"a"` (« append »). Ce mode fonctionne comme `"w"` dans le cas où le chemin pointe vers un fichier inexistant, mais n'écrase pas le fichier s'il existe. Dans ce dernier cas, le contenu du fichier n'est pas modifié et la position courante est initialisée à la fin du fichier. Par exemple, si `filename` pointe vers un fichier contenant une ligne `"Hello\n"`, l'exécution des instructions suivantes y ajoutent une seconde ligne `"word!\n"` :

```
with open(filename, "a", encoding="utf-8") as f: # Ouverture
    pour ajout.
  f.write("world!\n")
```

— Un fichier texte se comporte un peu comme une `str` dans le sens où, à part pour un ajout en toute fin de fichier, on ne peut pas directement ajouter ou supprimer de contenu dans un fichier texte. Pour effectuer ce genre d'opérations, le plus simple consiste souvent à créer nouveau fichier texte avec le contenu mis-à-jour voulu.

Exercice 189 (Modification de fichier, ⋆⋆)

1. *Écrire une fonction* `lower_file` *prenant en argument une chaîne de caractères* `filename` *supposée représentant le chemin d'un fichier texte existant et modifiant ce fichier en convertissant toutes ses lignes en bas de casse.*
2. *Écrire une fonction* `file_to_lower` *prenant en argument deux chaînes de caractères* `in_filename` *et* `out_filename` *supposées représentant deux chemins distincts (et d'un fichier texte existant pour* `in_filename`*), et créant à* `out_filename` *un fichier texte ayant pour contenu toutes les lignes de* `in_filename` *après conversion en bas de casse. Ne pas construire en mémoire tout le contenu de l'un ou l'autre des fichiers.*

□

Pour aller plus loin [Cours]

— Si l'on appelle la fonction `os.mkdir` avec comme argument un chemin pointant vers un dossier fictif mais situé dans un dossier existant, alors le dossier pointé par le chemin est créé (avec un contenu vide).
— Si l'on appelle la fonction `os.mkdir` avec comme argument un chemin pointant vers un élément existant, ou vers un dossier fictif situé dans dossier lui-aussi fictif, alors Python lève une exception.
— Si l'on appelle la fonction `os.makedirs` avec comme argument un chemin pointant vers un dossier fictif, alors le dossier pointé par le chemin, ainsi que tous ses dossiers parents non encore existants, sont créés.
— Si l'on appelle la fonction `os.makedirs` avec comme argument un chemin pointant vers un élément existant, alors Python lève une exception.

— Par exemple, si le système de fichiers contient à sa racine un dossier `"home"`, contenant un dossier vide `"guest"`, alors les deux suites d'instructions suivantes sont équivalentes :

```
os.mkdir("/home/guest/my_project")
os.mkdir("/home/guest/my_project/data")
```

```
os.makedirs("/home/guest/my_project/data")
```

— Le module `urllib` contient des fonctions permettant de récupérer de l'information à partir d'une URL (une chaîne de caractères que, en première approximation, l'on peut interpréter comme l'adresse d'un fichier sur internet). On peut accéder à ces fonctions après avoir exécuté l'instruction suivante :

```
import urllib
```

— En particulier, si l'on passe une URL comme argument à la fonction `urllib.request.urlopen` alors l'on obtient un objet (de type `http.client.HTTPResponse`) possédant une méthode `read` qui, elle, retourne l'information obtenue à l'URL indiquée (sous forme d'un objet de type `bytes`, c.-à-d. une séquence d'*octets*). Par exemple :

```
url = "https://www.gutenberg.org/cache/epub/14155/pg14155.txt"
with urllib.request.urlopen(url) as response:
  print(type(response)) # Affiche "<class 'http.client.
   HTTPResponse'>".
  content = response.read()
  print(type(content)) # Affiche "<class 'bytes'>".
```

— Un objet de type `bytes` peut être écrit dans un fichier ouvert avec le mode `"wb"` (« write binary ») et sans spécifier d'encodage (car on va y écrire directement une séquence de bits sans chercher à l'interpréter comme une chaîne de caractères). Par exemple :

```
url = "https://www.gutenberg.org/cache/epub/14155/pg14155.txt"
with urllib.request.urlopen(url) as response:
  with open("pg14155.txt", "wb") as f: # Ouverture pour é
   criture binaire.
    f.write(response.read())
```

— Un objet de type `bytes` qui représente du texte peut aussi être directement converti en un objet de type `str` grâce à la méthode `decode`. Par défaut, cette méthode suppose que le `bytes` suit l'encodage de caractères `UTF-8`. Par exemple :

```
url = "https://www.gutenberg.org/cache/epub/14155/pg14155.txt"
with urllib.request.urlopen(url) as response:
  text = response.read().decode()
  print(type(text)) # Affiche "<class 'str'>".
  print(len(text)) # Affiche le nombre de caractères dans `
   text`.
  with open("pg14155.txt", "w", encoding="utf-8") as f:
    f.write(text)
```

— Si un `bytes` encode un texte en suivant un autre format d'encodage, il est possible de le spécifier comme argument à `decode` (avec par exemple `"iso-8859-1"` plutôt que la valeur par défaut `"utf-8"`). Par exemple :

```
url = ... # URL d'un fichier encodé en ISO-8859-1.
with urllib.request.urlopen(url) as response:
  text = response.read().decode("iso-8859-1")
  with open("pg14155.txt", "w", encoding="iso-8859-1") as f:
    f.write(text)
```

8.4 Exercices supplémentaires

Exercice 190 (Parcours d'une branche, ⋆⋆⋆)
Écrire une fonction `listdir_rec` *prenant en argument une chaîne de caractères* `path` *et (i) retournant* `True` *et affichant les chemins de tous les fichiers descendants du dossier pointé par* `path` *si un tel dossier existe, et (ii) retournant* `False` *(sans rien afficher) sinon. La fonction doit être récursive, c.-à-d. que son fonctionnement repose sur le fait qu'un appel à* `listdir_rec` *apparaît dans le corps de* `listdir_rec` *elle-même.* □

Exercice 191 (Premières lignes, ⋆⋆)

1. *Écrire une fonction* `f` *prenant en argument une chaîne de caractères* `filename`*, supposée être un chemin vers un fichier texte, et un entier positif* `n`*, et affichant (sans saut de ligne supplémentaire) les* `n` *premières lignes de ce fichier (ou moins si le fichier en contient moins).*
2. *Modifier la fonction de telle sorte que si l'argument* `n` *passé est strictement négatif, toutes les lignes soient lues.*

□

Exercice 192 (Création de chemins par jointure (I), ⋆⋆)
Dans cet exercice, on cherche à implémenter des fonctions implémentant de manière plus ou moins simplifiée le comportement de `os.path.join`*.*

1. *Écrire une fonction* `myJoin` *prenant en argument une liste de chaînes de caractères* `l` *et retournant la chaîne obtenue en concaténant, séparés par des occurrences de* `os.sep`*, les élements non vides de* `l`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*.*

   Contrat :
   (En supposant que `os.sep = "/"`*.)*
   `l = ["..", "documents"]` → *retour :* `"../documents"`
   `l = ["a/b", ".", "c"]` → *retour :* `"a/b/./c"`
   `l = ["..", "", "documents", ""]` → *retour :* `"../documents"`
   `l = ["..", "", "documents", "", ""]` → *retour :* `"../documents"`

2. *Récrire* `myJoin` *de manière à ce que si un élément de l'argument* `l` *vaut ou commence par* `os.sep`*, tous les éléments précédents soient ignorés. Veiller à bien généraliser le comportement illustré dans le contrat, impliquant probablement de gérer les éléments égaux à* `os.sep` *différement des autres éléments commençant par* `os.sep`*.*

   Contrat :
   (En plus du contrat de la question précédente et toujours en supposant que `os.sep = "/"`*.)*
   `l = ["/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["/home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/", "home", "guest"]` → *retour :* `"/home/guest"`
   `l = ["..", "/", "tmp", "/home", "guest"]` → *retour :* `"/home/guest"`

□

Exercice 193 (Comptage d'occurrences, ⋆⋆)
Écrire une fonction `words_count` *prenant en argument une chaîne de caractères* `filename` *supposée être un chemin pointant vers un fichier texte, et retournant le dictionnaire associant à chaque mots apparaissant dans ce fichier son nombre d'occurrences. (Supposer que tous les tokens sont séparés par des espaces.)* □

Exercice 194 (Découpage, ⋆⋆⋆)
Écrire une fonction `split` *réimplémentant la méthode du même nom pour les séparateurs constitués d'un unique caractère, c.-à-d. prenant en argument une chaîne de caractères* `s` *et un caractère* `sep`*, et retournant la liste de chaînes de caractères obtenue en découpant* `s` *au niveau de chaque occurrence de* `sep`*. La fonction ne doit pas utiliser la méthode* `split`*. (Bien vérifier sur machine que la fonction proposée satisfait le contrat.)*

Contrat :
`s, sep = "abc.de..fgh.", "."` → *retour :* `["abc", "de", "", "fgh", ""]`
`s, sep = "abc.de..fgh", "."` → *retour :* `["abc", "de", "", "fgh"]`

□

Exercice 195 (Création de chemins par jointure (II), ⋆⋆⋆)
Écrire une fonction `myJoin` *réimplémentant* `os.path.join`*, à l'unique différence près qu'alors que* `os.path.join` *accepte n'importe nombre d'arguments de type* `str`*,* `myJoin` *doit accepter un unique argument* `l`*, une liste de chaînes de caractères. Attention,* `os.path.join` *a un comportement qui n'est pas facile à décrire ; étudier attentivement le contrat et ne pas hésiter à faire d'autres tests sur machine pour comprendre précisément la valeur retournée par* `os.path.join`*. La fonction ne doit pas utiliser* `os.path.join`*, mais peut utiliser* `join`*. Indice : Construire une liste de chaînes de caractères* `tmp` *et retourner* `"".join(tmp)`*.*

Contrat :
(En supposant que `os.sep = "/"`*.)*

`l = ["..", "documents"]`	→	*retour :* `"../documents"`
`l = ["a/b", ".", "c"]`	→	*retour :* `"a/b/./c"`
`l = ["..", "", "documents", ""]`	→	*retour :* `"../documents/"`
`l = ["..", "", "documents", "", ""]`	→	*retour :* `"../documents/"`
`l = ["/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/", "home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home/", "guest"]`	→	*retour :* `"/home/guest"`
`l = ["..", "/", "tmp", "/home//", "guest"]`	→	*retour :* `"/home/guest"`

□