CNESC Informatique

Python

Cours 2CB

2020–2021

Versions utilisées pour les tests
Python 3.8.3, 3.7.7, 3.6.1
Thonny 3.2.7

Tout le cours fonctionne également sur iPad, avec Pythonista 3.3

Auteurs de la version originale
Ben Kremer, David Mancini, Nino Silverio, Pascal Zeihen, François Zuidberg

à partir de la version 2.0
Ben Kremer, Pascal Zeihen, François Zuidberg

Version 2.1 – Luxembourg, le 3 juillet 2020

Table des matières

1 Introduction

Remarque préliminaire

Dans ce manuel, quelques sections et une partie des exercices sont marqués par des astérisques. Les exercices **sans astérisque** doivent tous être traités, soit en classe, soit par l'élève comme devoir à domicile.

Un astérisque * indique que la section du cours traite une matière plus technique ou plus difficile. Quant aux exercices, **un astérisque *** indique que l'enseignant est libre de les traiter en classe ou non et qu'il peut faire un choix parmi eux, sachant que le niveau de difficulté de ces exercices est le niveau à atteindre en fin d'année scolaire.

Deux astérisques ** indiquent que l'exercice en question est destiné aux élèves les plus motivés ; il ne sera probablement pas possible de le traiter en classe, faute de temps.

1.1 Un petit programme en Python

Voici un petit programme, qui montre de façon concise la beauté, la clarté et la puissance de Python. Il s'agit de résoudre une équation du second degré de la forme $ax^2+bx+c=0$ qui peut avoir soit deux solutions réelles ($\Delta > 0$), soit une solution réelle ($\Delta = 0$), soit aucune solution réelle ($\Delta < 0$).

```
# solve the quadratic equation a * x**2 + b * x + c = 0

# import square root method from math module
from math import sqrt

# let the user enter the real parameters
a = float(input("Enter a: "))
b = float(input("Enter b: "))
c = float(input("Enter c: "))

# calculate delta
d = b ** 2 - 4 * a * c

# find the solutions and output them to the screen
if d > 0:
    sol1 = (-b - sqrt(d)) / (2 * a)
    sol2 = (-b + sqrt(d)) / (2 * a)
    print("The solutions are", sol1, "and", sol2)
elif d == 0:
    sol = -b / (2 * a)
    print("The solution is", sol)
else:
    print("There are no real solutions.")
```

L'exemple ci-dessus montre quelques atouts du langage Python. Il est

- ⋆ simple
- ⋆ clair
- ⋆ modulaire et extensible

1.2 Qu'est-ce qu'un programme ?

Un programme est une suite de commandes pour résoudre un problème, par exemple un calcul mathématique comme la résolution d'un système d'équations ou la recherche des racines d'un polynôme. Un programme peut aussi manipuler des données non numériques, comme la recherche d'un mot dans un texte, le remplacement d'une partie d'un texte par une autre, l'enlèvement des yeux rouges sur une photo, etc. Chaque langage de programmation offre (entres autres) les concepts suivants :

- ⋆ entrée de données par clavier, fichier et autre,
- ⋆ sortie de données sur l'écran (affichage) ou dans un fichier,
- ⋆ expressions et calculs mathématiques,
- ⋆ structure alternative (if – then – else),
- ⋆ structures répétitives (for, while).

1.2.1 Langages de programmation

Comme les « langages naturels » (le français, l'anglais, ...), les langages de programmation sont des systèmes d'expression et de communication. Mais les « phrases » utilisées par ces langages, appelées programmes, forment des textes destinés à être compris par des ordinateurs. Cependant, les langages utilisés pour communiquer avec les ordinateurs ne sont pas tous considérés comme des langages de programmation. Il s'agit seulement des langages qui, théoriquement, sont suffisamment universels pour exprimer tous les traitements possibles (algorithmes) qu'un ordinateur peut effectuer. Ne sont pas considérés comme des langages de programmation les langages dits « de quatrième génération », conçus pour résoudre des problèmes spécifiques, comme l'interrogation de bases de données avec SQL (*Structured Query Language*) ou encore le calcul formel mathématique avec *Mathematica*, *Maple* ou *MatLab*. On peut considérer qu'un programme est un texte dépourvu de toute ambiguïté et qui doit être écrit en respectant scrupuleusement les « règles de grammaire » du langage.

1.2.2 Langages compilés et langages interprétés

Les programmes écrits en langage compilé sont traduits une fois pour toute en langage machine par un *compilateur* ; on obtient ainsi un fichier exécutable (extension .exe sous *Windows* ou .app sous macOS). Dans un langage compilé, le programme est directement exécuté sur l'ordinateur, donc il sera en général plus rapide que le même programme dans un langage interprété. Les programmes écrits en langage interprété ont besoin d'un programme appelé *interpréteur* pour être exécutés, l'interprétation se faisant au fur et à mesure de l'exécution. Dans un langage interprété, le même code source pourra marcher directement sur tout ordinateur. Avec un langage compilé, il faudra (en général) tout recompiler à chaque fois, et ce séparément pour chaque plateforme (système d'exploitation comme *Windows* ou macOS) ce qui pose parfois des soucis.

1.3 *Debugging* – la recherche des erreurs

Lors de la programmation, on produit des fautes qui sont appelées bug (bogue) et la méthode de trouver les bugs est le debugging (débogage). Il existe trois sortes de fautes :

1.3.1 Syntax Error

Python ne peut exécuter un programme que si la syntaxe est correcte, sinon l'interpréteur affiche un message d'erreur. La syntaxe comporte la structure d'un programme et les règles portant sur cette structure.

1.3.2 Runtime Error

Les erreurs de ce type n'apparaissent que si le programme est exécuté, on les appelle aussi *exceptions*, car il se passe un événement exceptionnel non prévu par le programmeur.

1.3.3 Semantic Error

Un programme qui contient une erreur sémantique peut être exécuté et utilisé sans affichage de messages d'erreur, mais le programme ne fait pas ce qu'on a voulu qu'il fasse. Par exemple on veut calculer la somme de deux nombres mais au lieu du symbole + on a utilisé le symbole *. Le programme fait alors exactement ce qu'on a programmé, mais on n'a pas programmé ce qu'on voulait en fait avoir.

1.4 Présentation de Python

Python est un langage de programmation interprété qui fonctionne en mode console et en mode programme. Il est entièrement gratuit et existe actuellement pour tous les systèmes d'exploitation majeurs comme p. ex. *Windows*, macOS, GNU/Linux, etc..

La première version est sortie en 1991. Créé par Guido van Rossum, il a voyagé du Macintosh de son créateur, qui travaillait à cette époque au Centrum voor Wiskunde en Informatica aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la *Python Software Foundation* (PSF), créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques des « Monty Python ».

1.4.1 Utilité de Python

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir tout au long de ce cours. Il est, en outre, très facile d'étendre les fonctionnalités existantes. Ainsi, il existe ce qu'on appelle des bibliothèques qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Avec Python on peut faire :

- ⋆ de petits programmes très simples, appelés scripts, chargés d'une mission très précise sur votre ordinateur ;
- ⋆ des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie. . . ;
- ⋆ des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- ⋆ créer des interfaces graphiques ;
- ⋆ faire circuler des informations au travers d'un réseau ;

⋆ dialoguer d'une façon avancée avec votre système d'exploitation ;

Bien entendu, vous n'allez pas apprendre à faire tout cela en quelques minutes. Mais ce cours vous donnera des bases suffisamment larges pour développer des projets qui pourront, par la suite, prendre de l'amplitude.

1.4.2 Un langage de programmation interprété

Python est un langage de programmation interprété, c'est-à-dire que les instructions sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages compilés » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « compilation ». À chaque modification du code, il faut relancer la compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous *Windows* que sous Linux ou macOS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété, bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre le code.

1.4.3 L'environnement de programmation Python

Afin de pouvoir travailler avec Python, il nous faut créer/installer un environnement de travail/programmation. L'environnement de travail consiste en quatre éléments principaux, qui sont

⋆ l'interpréteur Python, la base du langage,

⋆ un *debugger* (débogueur, chasseur d'erreurs), permettant de regarder « derrière la façade »,

⋆ des librairies, offrant des fonctionnalités préprogrammées, et

⋆ un éditeur confortable, qui nous met tous ces éléments à disposition.

Initialement Python est équipé de l'environnement IDLE, qui est suffisant mais peu confortable. Il existe beaucoup d'alternatives et la CNESC Informatique a décidé de se limiter à deux solutions, PyCharm Edu (`jetbrains.com/pycharm-edu/`) et Thonny (`thonny.org`). Les instructions d'installation et les liens se trouvent aussi sur le site de la CNESC Informatique (`edupython.script.lu`).

1.4.4 Les deux modes de Python

L'interpréteur - interactive mode

L'interpréteur se présente par le triple chevron `>>>` qui est l'invite de Python (*prompt* en anglais) et qui signifie que Python attend une commande. L'esprit d'utilisation de l'interpréteur est un peu le même que celui d'une calculatrice.

```
>>> 2+3*5
 17
```

L'interpréteur permet

- ⋆ de tester du code au fur et à mesure qu'on l'écrit ;
- ⋆ d'effectuer des calculs.

Attention : Un nombre décimal s'écrit avec un point et non une virgule. Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs, c'est pourquoi on préférera, dans la mesure du possible, travailler avec des nombres entiers.

L'éditeur - script mode

En mode interactif, Python affiche les résultats des expressions entrées. En mode éditeur, ceci ne peut être fait automatiquement. Pour voir les résultats il faut utiliser l'expression `print()` qui prend une liste de valeurs et qui imprime leurs représentations sous forme de texte dans le fichier *standard output file*. Le contenu de ce fichier sera alors affiché directement dans votre environnement de programmation.

1.5 Exercices

Exercice 1.1 – interactive mode

Pour se familiariser avec Python, effectuer les manipulations suivantes dans la fenêtre de l'interpréteur (Python Console) de l'environnement de programmation. Il n'est pas nécessaire d'utiliser la méthode `print()`, car les résultats seront affichés automatiquement.

1. Premiers pas avec les calculs :
   - ⋆ la somme de 27 et de 15
   - ⋆ la différence de 26 et de 18
   - ⋆ le produit de 6 par 7
   - ⋆ le quotient de 1024 par 512
   - ⋆ 247 divisé par 60 (= quotient de la division euclidienne)
   - ⋆ 247 modulo 60 (= reste de la division euclidienne)
2. Premiers pas avec les variables (entrer une ligne et taper sur Enter et entrer ensuite la ligne suivante etc.)

```
a = 2
b = 9
a + b
a * b
c = 42 * a
c + b
```

Exercice 1.2 – editor mode

Créer un nouveau programme, avec le nom `Ex_1_2.py`, qui contient le code-source suivant. Ensuite démarrer le programme et regarder ce qu'il produit comme affichage.

```
s1 = "Hello"
s2 = "world!"
print(s1, s2)
a = 2
b = 42
print(a + b)
name = input("Enter␣your␣name:␣")
print("Hello", name)

number1 = int(input("Enter␣a␣first␣number:␣"))
number2 = int(input("Enter␣a␣second␣number:␣"))
print(number1 * number2)
```

2 Notions de base

2.1 Variables

Une variable est un espace mémoire dans lequel il est possible de mettre une valeur. Par exemple, si en français on dit x est égal à 1, on utilise la variable dont le nom est x pour lui fixer la valeur 1. Pour faire la même chose en Python, on note simplement : `x = 1`.

Cette opération est appelée « affectation », et consiste à stocker une valeur en mémoire vive de l'ordinateur. On dit donc que l'on procède à l'affectation de la variable x avec la valeur 1.

En Python le symbole = est l'opérateur d'affectation. Cette valeur restera accessible jusqu'à la fin de l'exécution du programme (affichable avec `print(x)`).

La valeur de la variable peut être un littéral ou bien une expression (ex. : une autre variable ou une instruction comme `1+1`). L'expression est évaluée avant d'être affectée à la variable.

2.2 Types de valeurs

type		exemple
`int`	entier	`a = 5`
`float`	virgule flottante	`c = 5.6`
`str`	chaîne de caractères	`e = "hello"`
`list`	liste	`my_list = [23, "Joe", 4.5, "b"]`
`tuple`	tuplet	`my_tuple = ("a", 2.4, 45, "hello")`
`dict`	dictionnaire	`my_dict = {"name": "John", "age": 42}`

2.3 Différents opérateurs

2.3.1 Définitions

Un opérateur est un symbole utilisé pour effectuer un calcul entre des opérandes. Un opérande est une variable ou un littéral ou bien une expression. Une expression est une suite valide d'opérateurs et d'opérandes. Par exemple, dans l'expression $x = y + 1$, il y a deux opérateurs ($=$ et $+$) et trois opérandes (x, y et 1).

Il existe différents types d'opérateurs :

- ⋆ les opérateurs d'affectation
- ⋆ les opérateurs mathématiques
- ⋆ les opérateurs logiques
- ⋆ les opérateurs de comparaison
- ⋆ les opérateurs sur les séquences
- ⋆ les opérateurs numériques

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

2.3.2 Opérateurs d'affectation

```
x = 42            # simple assignment

x = y = z = 42    # multiple assignment

x, y = 42, 0.3    # parallel assignment
```

2.3.3 Opérateurs mathématiques

symbole	effet	example	result
`+`	addition	`6 + 4`	10
−	soustraction	`6 − 4`	2
*	multiplication	`6 * 4`	24
`/`	division	`6 / 4`	1.5
**	puissance	`6 ** 4`	1296
`//`	quotient de la division entière	`6 // 4`	1
		`−6.5 // 4.1`	−2.0
`%`	reste positif de la div. entière	`6 % 4`	2
		`−6.5 % 4.1`	1.7

On peut combiner l'opérateur d'affectation avec les opérateurs mathématiques : Par exemple, `x += 3` est une notation compacte pour dire `x = x + 3`, et de même `y /= z` correspond à `y = y / z` (attention à l'ordre implicitement choisi pour les opérations non commutatives !).

2.3.4 Opérateurs logiques

Les expressions avec un opérateur logique sont évaluées à **`True`** ou **`False`**.

- ⋆ X **`or`** Y : **ou** logique, si X est évalué à **`True`**, alors l'expression est **`True`** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- ⋆ X **`and`** Y : **et** logique, si X est évalué à **`False`**, alors l'expression est **`False`** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- ⋆ **`not`** X : **non** logique, évalué à la valeur booléenne opposée de X.

X	Y	X **and** Y	X **or** Y
False	**False**	**False**	**False**
False	**True**	**False**	**True**
True	**False**	**False**	**True**
True	**True**	**True**	**True**

X	**not** X
False	**True**
True	**False**

2.3.5 Opérateurs de comparaison

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **`True`** ou **`False`**. Les opérateurs de comparaison s'appliquent sur tous les types de base.

symbole	effet
<	strictement inférieur
>	strictement supérieur
<=	inférieur ou égal
>=	supérieur ou égal
==	égal
!=	différent de

Il est possible d'enchaîner les opérateurs : $X < Y < Z$, dans ce cas, c'est Y qui est pris en compte pour la comparaison avec Z et non pas l'évaluation de $(X < Y)$ comme on pourrait s'y attendre dans d'autres langages.

2.3.6 Priorité des opérateurs

1	**	puissance
2	−, +	signe
3	*, /, //, %	multiplication et division
4	+, −	addition et soustraction
5	==, <=, >=, <, >, !=	opérateurs de comparaison
6	**not**	logique : not
7	**and**	logique : and
8	**or**	logique : or

2.4 Opérateurs sur les chaînes de caractères (strings)

Les chaînes de caractères sont de type `str`. Comme délimiteurs de chaînes constantes, on peut soit écrire des apostrophes soit des guillemets, mais pour une même chaîne les deux délimiteurs doivent être identiques :

```
'une␣chaîne'
"une␣autre␣chaîne"
'Il␣dit␣:␣"Bonjour␣!"'
"Il␣dit␣:␣\"Bonjour␣!\""
"l'un␣et␣l'autre"
'l\'un␣et␣l\'autre'
```

Les trois derniers exemples illustrent que si l'on désire inclure dans la chaîne le symbole du délimiteur choisi, on doit le faire précéder d'un backslash \.

Les deux opérateurs simples applicables aux chaînes de caractères sont la concaténation + et la multiplication *.

```
"hello" + "world"  # 'helloworld'

"python" * 4       # 'pythonpythonpythonpython'
```

Dans les codes-sources de ce manuel, les strings sont toujours imprimés en vert.

2.5 Entrée de données - lecture du clavier

Il est très pratique de pouvoir demander à l'utilisateur de saisir une chaîne de caractères. Pour cela, Python dispose d'une instruction `input()`. Cette instruction renvoie une chaîne de caractères. Ainsi, si l'utilisateur tape le mot `Luxembourg`, le résultat stocké est `Luxembourg`.

```
>>> my_word = input()
  Luxembourg
>>> my_word
'Luxembourg'
```

Pour faciliter l'usage d'un tel programme, on peut passer un texte à afficher comme argument à l'instruction `input()` :

```
>>> country = input("Enter␣the␣name␣of␣a␣country:␣")
  Enter the name of a country: Belgium
>>> country
'Belgium'
```

Pour utiliser des nombres entrés par un utilisateur, on doit convertir la chaîne de caractères en un type numérique. Pour la conversion de la chaîne de caractères en un nombre entier, on utilise la fonction `int()` :

```
>>> number = int(input("Enter␣a␣number␣:␣"))
  Enter a number : 12
>>> number + 13
25
```

Voici quelques exemples :

```
# Enter a string without prompt
s = input()

# Enter a string with prompt
t = input("Enter␣a␣string:␣")

# Enter an integer with prompt (error if bad entry)
n = int(input("Enter␣an␣integer␣number:␣"))

# Enter a float with prompt (error if bad entry)
m = float(input("Enter␣a␣float␣number:␣"))
```

Un programme qui utilise ces instructions se présente alors comme suit :

```
product_name = input("Please␣enter␣the␣product␣name␣:␣")
product_price = int(input("Please␣enter␣the␣product␣price␣:␣"))
quantity = int(input("Please␣enter␣the␣quantity␣:␣")

price_to_pay = product_price * quantity
```

2.6 Affichage des données – editor mode

Les affichages de texte (*strings, chaînes de caractères* en Python sont réalisés à l'aide de l'instruction `print(text)` : Python affiche le texte et passe à une nouvelle ligne. Si on veut

terminer l'affichage avec un autre caractère ou un autre texte au lieu d'une nouvelle ligne, on peut utiliser une instruction comme `print(text, end = ".")` : Le texte est affiché et suivi de ce qui est défini par l'option `end`. Il existe encore d'autres opérations utiles qui sont possibles avec l'instruction `print()`, illustrées dans les exemples suivants :

```
# print nothing, followed by a new line
print()
```

```
>>>
```

```
# the string "Hello" is printed, followed by a new line
print("Hello")
```

```
>>> Hello
>>>
```

```
# print both strings with a blank space "Hello World", followed by a new line
print("Hello", "World")
```

```
>>> Hello World
>>>
```

```
# concatenate two strings, followed by a new line
print("Hello" + "World")
```

```
>>> HelloWorld
>>>
```

```
# print a string and a number (no conversion needed), followed by a new line
print("Hello", 42)
```

```
>>> Hello 42
>>>
```

```
# print a string and a number (with conversion), followed by a new line
print("Hello" + str(42))
```

```
>>> Hello42
>>>
```

```
# print the value from a variable, followed by a new line
my_number = 42
print("My␣number␣is", my_number)
```

```
>>> My number is 42
>>>
```

```
# print a string several times, followed by a new line
print(3 * "Hello")
```

```
>>> HelloHelloHello
>>>
```

```
# replace 'new line' by any other character(s)
print("Hello", end = "my dear!")
```

```
>>> Hello my dear!
```

```
# replace 'new line' by any other character(s)
print("We", end = " <3 ")
print("Python")
```

```
>>> We <3 Python
>>>
```

2.6.1 Formatted Strings

On peut afficher un mélange de texte et de contenu de variables comme dans les exemples précédents :

```
my_number = 42
print("My number is", my_number)
```

Ce qui est nouveau depuis la version 3.6 de Python, ce sont les chaînes de caractères formatées. Une chaîne de caractères formatée littérale ou f-string est une chaîne de caractères littérale préfixée par `f` ou `F`. Ces chaînes peuvent contenir des champs à remplacer, c'est-à-dire des expressions délimitées par des accolades {}. Alors que les autres littéraux de chaînes ont des valeurs constantes, les chaînes formatées sont de vraies expressions évaluées à l'exécution. Dans ce chapitre on illustre les deux méthodes et dans les chapitres suivants, on utilisera le plus souvent la méthode des chaînes de caractères formatées (f-string).

Exemples d'application :

```
my_number = 42

print(f"My number is {my_number}.")     # 'My number is 42.'
```

```
name = "Aline"
age = 37

print(f"Hello {name}! You are {age}.")     # 'Hello Aline! You are 37.'
```

```
a = 2
b = 3

print(f"{a} times {b} is {a*b}")     # '2 times 3 is 6'
```

2.7 Importations de modules

Un module est une sorte de bibliothèque (un regroupement de fonctions prédéfinies) qui, une fois importée, permet d'accéder à de nouvelles fonctions. Pour importer un module on utilise l'instruction **import** tout au début du programme. Il existe différentes approches pour importer les modules :

```
# import a module
import math           # recommended

print(math.sqrt(9))

# or import everything in the module
from math import *    # not recommended (many useless identifiers)

print(sqrt(9))

# or import only what you really need
from math import sqrt, cos, sin, pi

print(sqrt(9))
print(cos(pi))
print(sin(30))
```

Pour une liste complète des possibilités offertes par ce module, veuillez vous référencer à `docs.python.org` (Python module index) ou aux liens sur le site du cours.

2.7.1 math package et fonctions mathématiques

Le module `math` est un module qui permet d'avoir accès aux fonctions mathématiques comme le cosinus (`cos`), le sinus (`sin`), la racine carrée (`sqrt`), le nombre π (`pi`) et bien d'autres. Un module peut être importé partiellement ou complètement :

```
# import some math functions
# abs(), round() always available without import
from math import sin, cos, tan, pi, sqrt, ceil, trunc, floor

x = cos(pi)       # -1.0
x = sin(30)       # -0.9880316240928618 (arg in radians!)
x = abs(-3)       #  3
x = sqrt(256)     # 16
x = ceil(1.23)    #  2      (smallest integer larger or equal)
x = trunc(-8.76)  # -8      (strip fractional part)
x = floor(-8.76)  # -9      (largest integer smaller or equal)
x = round(4.6)    #  5      (rounds to nearest integer)
x = round(3.5)    #  4      (rounds to nearest EVEN integer!!)
x = round(4.5)    #  4      (rounds to nearest EVEN integer!!)
```

2.7.2 Nombres pseudoaléatoires : random package

En anglais « random » signifie « le hasard ». Ce module permet d'utiliser des fonctions générant des nombres aléatoires.

```
from random import random, randint, randrange

x = random()              # random float 0.0 <= x < 1.0
x = randrange(6)          # random int   0 <= x < 6
x = randrange(1, 6)       # random int   1 <= x < 6
x = randrange(1, 18, 4)   # random int is one of 1, 5, 9, 13, 17
x = randint(1, 6)         # random int   1 <= x <= 6
```

2.8 Divers

2.8.1 Commentaires

```
# voici un commentaire d'une ligne
```

```
'''commentaire sur
plusieurs lignes
dans le code source
'''
```

```
"""commentaire sur
plusieurs lignes
dans le code source
"""
```

Les commentaires à plusieurs lignes sont en fait des **chaînes de caractères** à plusieurs lignes, dont les délimiteurs sont triplés et qui ne servent à rien lors de l'exécution du programme.

Les IDE (*Integrated Development Environments*) utilisés (PyCharm, Thonny) permettent de placer en commentaire plusieurs lignes ou de les sortir d'un commentaire d'un seul coup : pour Thonny p. ex., il suffit de sélectionner les lignes en question et de choisir *Comment out* ou *Uncomment* dans le menu *Edit*.

Quelques recommandations :

- ⋆ Écrire un commentaire au début de chaque fichier-source, pour expliquer en quelques mots ce que le programme est censé faire.
- ⋆ Écrire un commentaire au début ou à la fin de chaque fichier-source, pour donner les références lorsqu'on utilise des parties de code ou des algorithmes sous copyright ou copiés/collés du Web.
- ⋆ Écrire un commentaire par bloc de code non définitif, pour préciser quel type de maintenance est requis : faut-il corriger ou compléter le code, faut-il distinguer encore des cas particuliers ? Cela est notamment utile lors des devoirs en classe et des examens : guider le correcteur dans sa tâche rapporte généralement plus de points que toute tentative d'obstruction, de bluff ou de code-bidon.
- ⋆ Écrire un commentaire pour chaque déclaration d'une nouvelle classe, pour expliquer brièvement à quoi elle sert.
- ⋆ Écrire un commentaire pour chaque fonction, pour en préciser le fonctionnement ou les algorithmes utilisés. Exception : si le code de la fonction est assez triviale et son nom est bien choisi, le commentaire peut être omis.

- ⋆ Écrire un commentaire pour chaque variable rendue globale dans une fonction afin d'expliquer pourquoi on a besoin de la variable extérieure et quel est l'effet éventuel subi par elle.
- ⋆ Écrire un commentaire pour chaque bloc de code que l'on juge particulièrement difficile ou astucieux.
- ⋆ Écrire un commentaire pour chaque bloc de code dont l'exécution dépend de la version de Python utilisée (p. ex. un code qui fonctionne bien sous Python 3.7, mais qui cause des problèmes sous Python 3.6).
- ⋆ Et surtout : ne pas écrire d'autres commentaires, qui risquent de détourner l'attention du lecteur.

2.8.2 Conversion de types

```
int(s)    # convert string s to integer
float(s)  # convert string to float
str(n)    # convert integer or float n to string
list(x)   # convert tuple, range or similar to list
```

2.9 Exercices

Exercice 2.1 Que va afficher la portion de code suivante ? Introduire le programme dans Python et vérifier le résultat !

```
print('hello world')
print('hello world', 6 * 7)
print("If I add", 4, "to", 20, "I'll get", 20 + 4)
print('hello' * 2)
```

Exercice 2.2 Que va afficher la portion de code suivante ? Introduire le programme dans Python et vérifier le résultat !

```
a=2
b=9
print(a+b)
c=a*b
print(c)
d=a-(b+c)**2
print(d)
```

Variante : Utiliser les f-strings et rajoutez du texte aux affichages.

Exercice 2.3 Écrire une instruction qui calcule et affiche le résultat de chacune des expressions suivantes :

1. $15 + 7 \cdot 4 + 2/4$
2. $17 - [14 - 3 \cdot (7 - 2 \cdot 8)] \cdot 2$
3. $(12 - 5) \cdot 7 - (13 + 3) * 4$

4. $2 - 7 + 2 \cdot (3 - 5) * 2$
5. $5^2 - 7^3 - 5^4$
6. $\sqrt{169} + 202$

Exercice 2.4 Effectuer (sans l'aide de l'ordinateur) les expressions suivantes :

1. `86 // 15`
2. `86 % 15`
3. `(5−(8−2*2)*2)*2`
4. `sqrt(9**2+19)`
5. `145−(145//13)*13`
6. `288 % 18`
7. `288 / 18`

Exercice 2.5 Écrire un programme qui demande à l'utilisateur d'entrer deux nombres entiers du clavier et les affecte respectivement aux variables `n1` et `n2`. Écrire ensuite des instructions pour afficher la somme et le produit de `n1` et de `n2`.

Exercice 2.6 Écrire un programme qui demande à l'utilisateur d'entrer deux nombres entiers du clavier et les affecte respectivement aux variables `n1` et `n2`. Écrire ensuite des instructions qui permettent d'échanger (*swap*) les valeurs des variables `n1` et `n2` :

- ⋆ Version 1 : utiliser une troisième variable temporaire
- ⋆ Version 2 : Python sait échanger très facilement deux variables (penser aux affectations parallèles).

Exercice 2.7 Écrire un programme qui lit trois nombres réels du clavier, calcule leur moyenne (arithmétique) et affiche le résultat.

Exercice 2.8 Écrire un programme qui lit trois nombres réels strictement positifs du clavier, calcule leur moyenne harmonique et l'affiche. Rappelons que la moyenne harmonique de n nombres a_i strictement positifs est donnée par la formule :

$$H = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_n}}$$

Exercice 2.9 Écrire un programme qui lit un nombre réel strictement positif qui représente le prix d'un article quelconque (TVA de 17% comprise, TTC). Le programme calculera le prix hors TVA (HTVA) de l'article et l'affichera.

Exemple d'exécution :

```
Enter your price (taxes included, TTC): 234
Price without taxes (HTVA): 200.0
```

Exercice 2.10 Écrire un programme qui lit du clavier quatre nombres réels strictement positifs, réprésentant 4 résistances d'un circuit en parallèle. Le programme calculera ensuite la résistance résultante par la loi d'Ohm et l'affichera.

$$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4}$$

Exercice 2.11 Écrire un programme qui lit trois nombres réels, représentant les mesures des 3 côtés d'un triangle. (Ces nombres doivent donc vérifier l'inégalité triangulaire, par exemple : $a = 6$, $b = 5$, $c = 4$). Le programme devra ensuite calculer, à l'aide de la formule d'Héron, l'aire du triangle correspondant à ces mesures et l'afficher. Cette formule permet de calculer l'aire d'un triangle quelconque en ne connaissant que les longueurs a, b et c de ses trois côtés.

Formule d'Héron : $a = \sqrt{s(s-a)(s-b)(s-c)}$ avec $s = \frac{a+b+c}{2}$

Exercice 2.12 Écrire un programme qui lit le dividende et le diviseur non nul d'une division et qui affiche la division euclidienne résultante.

Exemple : Si le dividende est 34 et le diviseur 8, le programme devra afficher `34 = 8 * 4 + 2`.

Exercice 2.13 – Suite arithmétique

Écrire un programme qui lit le premier terme (*first term*) et la raison (*common difference*) d'une suite arithmétique. Le programme lira ensuite un nombre naturel n, calculera le n-ième terme et l'affichera (En anglais : *arithmetic sequence with a common difference*).

Exemple d'exécution :

```
first term: 4
common difference: 2
n: 5
n-th number of sequence: 12
```

2.10 Styleguide

Pour rendre votre programme lisible pour d'autres développeurs et pour rester conformes aux conventions définies dans PEP8, veuillez respecter les directives suivantes :

- ⋆ Les opérateurs doivent être entourés d'espaces.
- ⋆ Un nom de variable est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre. Le symbole _ (underscore) est assimilé à une lettre.
- ⋆ Python se réserve les noms suivants (liste exhaustive) qui ne peuvent pas être utilisés comme noms de variable : **`and`**, **`as`**, **`assert`**, **`async`**, **`await`**, **`break`**, **`class`**, **`continue`**, **`def`**, **`del`**, **`elif`**, **`else`**, **`except`**, **`finally`**, **`for`**, **`from`**, **`global`**, **`if`**, **`import`**, **`in`**, **`is`**, **`lambda`**, **`nonlocal`**, **`not`**, **`or`**, **`pass`**, **`raise`**, **`return`**, **`try`**, **`while`**, **`with`**, **`yield`**, **`False`**, **`None`**, **`True`**

  Dans ce cours, ces mots réservés (*keywords*) sont toujours imprimés en bleu.
- ⋆ Python connaît en outre un certain nombre d'instructions, de fonctions, de types... qui sont **toujours** disponibles, sans qu'on ait besoin d'importer un module. Le programmeur peut redéfinir ces noms pour autre chose (p. ex. des variables), mais alors il risque de perturber, voire de perdre totalement la fonctionnalité initialement associée au nom. Voici la liste de ces noms, qu'on trouve dans la documentation officielle de Python : `abs`, `all`, `any`, `ascii`, `bin`, `bool`, `breakpoint`, `bytearray`, `bytes`, `callable`, `chr`, `classmethod`, `compile`, `complex`, `delattr`, `dict`, `dir`, `divmod`, `enumerate`, `eval`, `exec`, `filter`, `float`, `format`, `frozenset`, `getattr`, `globals`, `hasattr`, `hash`, `help`, `hex`, `id`, `input`, `int`, `isinstance`, `issubclass`, `iter`, `len`, `list`, `locals`, `map`, `max`, `memoryview`, `min`, `next`, `object`, `oct`, `open`, `ord`, `pow`, `print`, `property`, `range`, `repr`, `reversed`, `round`, `set`, `setattr`, `slice`, `sorted`, `staticmethod`, `str`, `sum`, `super`, `tuple`, `type`, `vars`, `zip`, `__import__`

  Dans ce cours, ces mots sont toujours imprimés en pourpre.

⋆ La « casse » est significative : les caractères majuscules et minuscules sont distingués. Ainsi `python, Python, PYTHON` sont trois variables différentes.

Les concepts décrits dans les paragraphes suivants seront introduits ultérieurement dans le cours de 2CB et celui de 1CB.

⋆ Lettres seules, en minuscule : pour les boucles et les indices.

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

i = get_index() + 12
print(my_list[i])
```

⋆ Lettres minuscules avec underscore : pour les modules, variables, fonctions et méthodes.

```
my_var = 10
def my_function():
    ...

class MyClass:
    def first_method(self):
        ...
```

⋆ Lettres majuscules et underscore : pour les (pseudo-)constantes.

```
MAX_SIZE = 100000  # à mettre après les imports
```

⋆ *Upper camel case* : nom de classe.

```
class ThisIsMyFirstClass:
    def first_method(self):
        ...
```

⋆ Si le nom contient un acronyme, on fait une entorse à la règle :

```
class HTMLParserCQFDDDTCCMB:
    def first_method(self):
        ...
```

⋆ On n'utilise **pas** le *lower camel case* (`myClass`) en Python (notation préférée en Swift par exemple) !

3 Structure alternative

3.1 Problème introductif : Fitness

La société de fitness **PyGym** propose des séances de gymnastique à raison de 3€/h. Pour un achat de plus de 20 heures, ce prix se réduit à 2,50€/h. On se propose de créer un programme Python qui, en partant du nombre d'heures achetées, affiche le ticket de caisse.

```
1  print("***␣Welcome␣to␣PyGym␣***")
2
3  # data input
4  hours = int(input("Please␣enter␣the␣number␣of␣hours␣:␣"))
5  print("\nThank␣You!\n")
6
7  # processing
8  price = 3.0
9
10 if hours > 20:
11     price = 2.5
12
13 total = hours * price
14
15 # data output
16 print("number␣of␣hours:␣%3d␣hours" % hours)
17 print("item␣price:␣␣␣␣␣␣%6.2f␣EUR" % price)
18 print("total:␣␣␣␣␣␣␣␣␣␣␣%6.2f␣EUR" % total)
19
20 input("\nPlease␣hit␣ENTER␣to␣quit...")
21
22 # VARIANTE avec f-strings
23 print("***␣Welcome␣to␣PyGym␣***")
24
25 # data input
26 hours = int(input("Please␣enter␣the␣number␣of␣hours␣:␣"))
27 print("\nThank␣You!\n")
28
29 # processing
30 price = 3.0
31
32 if hours > 20:
33     price = 2.5
34
35 total = hours * price
36
37 # data output
38 print(f"number␣of␣hours:␣{hours:3d}␣hours")
39 print(f"item␣price:␣␣␣␣␣␣{price:6.2f}␣EUR")
40 print(f"total:␣␣␣␣␣␣␣␣␣␣{total:6.2f}␣EUR")
41 input("\nPlease␣hit␣ENTER␣to␣quit...")
```

Les lignes 10 et 11 effectuent le travail désiré : SI hours > 20 ALORS price = 2.5.

Remarquer également le formatage des nombres affichés (partie après le double-point entre

accolades) dans les lignes 16–18 et, en cas d'intérêt, consulter l'aide Python officielle pour les options de formatage les plus utiles.

On peut donc prendre une décision sur base d'une condition logique. Une telle structure de programmation est appelée **structure alternative**. Python nous propose plusieurs variantes de structures alternatives. Celle utilisée ci-dessus est l'alternative simple.

La première ligne d'une structure alternative commence par le mot-clé **`if`**, suivie par une *condition logique* et se termine par un double-point (`:`). Au-dessous vient le bloc d'instruction(s) à exécuter. **Ce bloc doit être mis en retrait (indenté)**. En effet, Python délimite la portée de ses structures de contrôle par indentation.

Une condition est une expression logique qui donne donc comme résultat la valeur vrai (**`True`**) ou faux (**`False`**). Elle peut aller du plus simple au très complexe. Le plus souvent elle se compose de deux opérandes et d'un opérateur de comparaison. Plusieurs conditions peuvent être combinées par des opérateurs logiques. Tout comme les opérateurs arithmétiques, les opérateurs logiques sont eux aussi soumis à des règles de priorités. Référez-vous aux sections du chapitre précédent pour les détails.

3.2 Différentes variantes

3.2.1 Alternative simple

Cette variante teste si la condition donnée est vérifiée (donc si elle donne le résultat booléen **`True`**). Dans le cas affirmatif, le bloc d'instruction(s) indenté est exécuté. Dans le cas contraire (si résultat est **`False`**) le programme continue avec l'instruction suivante (non indentée).

```
if <condition>:
    <instruction(s)>
```

Exemple : un vendeur reçoit un bonus de 500€ si son chiffre d'affaires dépasse les 100 000€ :

```
bonus = 0
if sales > 100000:
    bonus = 500
print(bonus)
```

3.2.2 Alternative complète

On peut compléter la variante simple par un bloc d'instruction(s) à effectuer dans le cas où la condition à tester donne le résultat **`False`**. Dans ce cas on ajoute le mot-clé **`else:`** (non-indenté et suivi d'un double-point) et, au-dessous, le bloc d'instruction(s) indenté.

```
if <condition>:
    <instruction(s)>
else:
    <instruction(s)>
```

Exemple : À l'Olympiade Mathématique Belge, une bonne réponse est récompensée par 3 points, alors qu'une mauvaise réponse est sanctionnée par un retrait de 2 points :

```
if answer == correct_answer:
    score += 3
else:
    score -= 2
print(score)
```

N. B. Aucun bloc d'instructions ne peut être vide. Donc après un : il faut au moins une instruction. Lorsqu'on ne veut vraiment rien faire dans un bloc indenté, l'instruction-bidon **pass** peut servir.

3.2.3 Opérateur ternaire *

L'opérateur ternaire est le seul opérateur à trois arguments couramment utilisé dans les langages de programmation modernes. Il permet de réduire une structure alternative **if** - **else** d'au moins 4 lignes en une seule ligne. Voici l'exemple précédent (de l'Olympiade Mathématique Belge) exprimé à l'aide de l'opérateur ternaire :

```
score += 3 if answer == correct_answer else -2
```

L'expression devant **if** est évaluée lorsque la condition écrite entre **if** et **else** est vérifiée ; sinon c'est l'expression après **else** qui est évaluée. Cet opérateur fonctionne donc uniquement avec des expressions et ne permet pas d'exécuter conditionnellement des instructions quelconques.

Voici, en guise de 2e exemple, le calcul du bonus du vendeur, de la page précédente :

```
bonus = 500 if sales > 100000 else 0
```

En général :

```
my_var = <expr_if_true> if <condition> else <expr_if_false>
```

3.2.4 Alternative multiple

Cette variante permet de cascader ou imbriquer (allemand : *verschachteln)* plusieurs structures alternatives, ainsi la branche **else:** peut contenir une deuxième structure alternative. Dans ce cas on remplace les deux mots-clé **else:** et **if** par le mot-clé **elif** et on n'effectue pas d'indentation supplémentaire. Le nombre d'imbrications n'est limité que par la taille de mémoire mise à disposition.

```
if <condition_1>:
    <instruction(s)>
elif <condition_2>:
    <instruction(s)>
...
elif <condition_n>:
    <instruction(s)>
else:
    <instruction(s)>
```

Exemple : une note de devoir est à classer selon le tableau suivant :

note	mention
[1, 10[	très mauvais
[10, 20[	mauvais
[20, 30[	insuffisant
[30, 40[	suffisant
[40, 50[	bien
[50, 60]	très bien

```
if mark < 10:
    result = "very bad"
elif mark < 20:
    result = "bad"
elif mark < 30:
    result = "insufficient"
elif mark < 40:
    result = "sufficient"
elif mark < 50:
    result = "good"
else:
    result = "very good"

print(f"The mark {mark} is considered to be {result}.")
```

3.3 Exercices

Exercice 3.1 Créer un programme qui affiche la valeur absolue d'un nombre entré au clavier (sans utiliser la fonction `abs`).

Exercice 3.2 Créer un programme qui affiche le plus grand de trois nombres entiers entrés au clavier (sans utiliser la fonction `max`).

Exercice 3.3 Créer un programme qui vérifie si un triangle est rectangle en entrant les trois côtés au clavier.

Exercice 3.4 Créer un programme qui affiche les signes du produit et de la somme de deux nombres entrés au clavier (sans effectuer les opérations en question). Par abus de simplification, le nombre 0 est considéré ici comme positif (et non négatif).

Exercice 3.5 Créer un programme qui insère un nombre d dans une suite croissante de nombres a, b et c. À la fin il faut avoir $a \leq b \leq c \leq d$.

Exercice 3.6 Créer un programme qui trie trois nombres a, b et c entrés au clavier (sans utiliser la fonction *max* ou *min*). À la fin il faut avoir $a \leq b \leq c$.

Exercice 3.7 Créer un programme qui teste le calcul mental de l'utilisateur. Pour cela il calcule deux nombres entiers aléatoires $\in [10, 20]$, Ensuite il en demande la somme, la différence, le produit et le quotient entier. Chaque réponse correcte est confirmée par un « Correct ! » et chaque réponse fausse est corrigée par l'affichage de la bonne réponse. À la fin, le nombre de bonnes et mauvaises réponses est affiché.

Exercice 3.8 Créer un programme qui vérifie si une année a entrée au clavier est une année bissextile ou non. Une année bissextile (allemand : *Schaltjahr) ; anglais :* leap year comporte 366 jours au lieu de 365.

Règles à appliquer :

1. Si une année a est un multiple de 4, alors a est une année bissextile.
2. Exception à la règle 1 : Si a est un multiple de 100, alors a n'est pas une année bissextile.

3. Exception à la règle 2 : Si a est un multiple de 400, alors a est une année bissextile.

Ainsi 1904, 2000 et 2020 sont des années bissextiles, mais 1000, 1901 et 2019 n'en sont pas.

Exercice 3.9 Créer un programme qui affiche le BMI et son interprétation verbale pour une personne à partir de sa taille et de son poids.

Le BMI (simplifié pour nos besoins) se calcule comme suit :

$$\text{BMI} = \frac{\text{poids [kg]}}{(\text{taille [m]})^2}$$

L'interprétation verbale sort du tableau suivant :

BMI	**Weight Status**
[0, 18.5[	underweight
[18.5, 25[	healthy weight
[25, 30[	overweight
[30,∞[	obese

Exercice 3.10 Créer un programme qui vérifie si un nombre entier $n \in [1, 100]$ entré au clavier est premier ou non.

4 Boucles

4.1 Problème introductif

Le dernier exercice du chapitre précédent a permis de vérifier si un nombre entier $n \in [1, 100]$ entré au clavier était premier ou non. Pour cela il a fallu passer un certain nombre de tests (validité, puis égalité et divisibilité).

Une solution possible en est la suivante :

```
# information
print("*** Verify if a given number is prime ***\n")

# data input
n = int(input("Please enter a number [1..100] : "))

print("\nThank You!\n")

# processing
if 1 <= n <= 100:
    if n < 2:
        result = "not prime"
    elif (n != 2) and (n % 2 == 0):
        result = "not prime"
    elif (n != 3) and (n % 3 == 0):
        result = "not prime"
    elif (n != 5) and (n % 5 == 0):
        result = "not prime"
    elif (n != 7) and (n % 7 == 0):
        result = "not prime"
    else:
        result = "prime"
else:
    result = "not in the requested set of numbers!"

# data output
print(n, "is", result)

# the end
input("\nPlease hit ENTER to quit...")
```

On voit qu'après le test de validité de n, le programme est essentiellement formée de deux lignes de code qui se répètent et qui ne diffèrent que par le diviseur. Si on voulait traiter des nombres plus grands, on imagine aisément que non seulement le programme s'allongerait rapidement (en fait il faut tester davantage de diviseurs), mais qu'il devient aussi plus difficile à gérer (il ne faut ni se tromper d'indentation, ni des diviseurs, etc...).

Ce problème peut être résolu grâce à la structure répétitive (boucle). En effet, en utilisant une variable comme diviseur (au lieu de nombres constants) et en se servant d'une boucle, on rend ce programme plus court, plus élégant et plus convivial à lire et à éditer. En plus, comme on utilise une variable comme diviseur, on peut se passer de la limite supérieure pour le nombre n. L'approche change aussi en ce sens qu'on se sert du fait qu'un nombre premier n a exactement deux diviseurs différents $(1, n)$. Finalement, en commençant les tests de divisibilité par le nombre 1, le programme reste valable pour les nombres négatifs.

Le code source devient alors :

```
1  # information
2  print("*** Verify if a given number is prime ***\n")
3
4  # data input
5  n = int(input("Please enter a number: "))
6
7  print("\nThank You!\n")
8
9  # processing
10 number_of_divisors = 0
11 i = 1
12 while i <= n:
13     if n % i == 0:
14         number_of_divisors += 1
15     i += 1
16
17 if number_of_divisors == 2:
18     result = "prime"
19 else:
20     result = "not prime"
21
22 # data output
23 print(n, "is", result)
24
25 # the end
26 input("\nPlease hit ENTER to quit...")
```

Les lignes 10 à 15 réalisent le travail proprement dit :

- ⋆ Les lignes 10 et 11 initialisent les variables utilisées.
- ⋆ La ligne 12 contient l'instruction de boucle et son contrôle (condition d'arrêt).
- ⋆ Les lignes indentées 13, 14 et 15 constituent le corps de la boucle et sont exécutées à chaque itération (allemand : *Schleifendurchgang).*

On commence donc par $i = 1$, ensuite tant que $i \leq n$ on teste si i est un diviseur de n (avec, le cas échéant, incrémentation du compteur) puis on incrémente la variable i et on recommence.

Cela nous mène à la construction générale d'une boucle qui comporte les 4 étapes suivantes :

1. construction du corps de la boucle *(les instructions à répéter)*,
2. préparation de l'itération suivante *(faire évoluer les variables et/ou le contrôle)*,
3. définir le contrôle de la boucle *(la condition d'arrêt)*[a],
4. initialiser toutes les variables utilisées *(sur l'élément neutre, sur la valeur de départ, etc)*.

[a] Par abus de langage, le contrôle est appelé *condition d'arrêt*, alors qu'en réalité il s'agit de la ***condition d'exécution*** de la boucle !

Dans l'exemple ci-dessus :

- ⋆ les lignes 13 et 14 représentent le corps propre de la boucle,

⋆ la ligne 15 représente la préparation de l'itération suivante,

⋆ la ligne 12 représente le contrôle de la boucle,

⋆ les lignes 10 et 11 représentent l'initialisation de toutes les variables utilisées.

Il existe plusieurs types de boucles :

⋆ la boucle **while**,

⋆ la boucle **for**,

⋆ la boucle `repeat` (elle n'existe pas en Python, mais dans beaucoup d'autres langages).

Chaque type de boucle possède un certain nombre de caractéristiques et est prédestiné pour un certain type d'application. Parmi les différents types de boucles, la boucle **while** est la plus universelle. Chaque boucle de type **for** ou `repeat` peut être réécrite en boucle de type **while**, alors que le contraire n'est pas toujours possible.

4.1.1 La boucle while

La boucle de type **while** est la plus universelle. Est est utilisée lorsqu'on ne connaît pas d'avance le nombre d'itérations (donc de répétitions). Sa syntaxe est la suivante :

```
while <condition>:
    <instruction(s)>
```

La condition (le contrôle) n'est rien d'autre qu'une expression logique donnant le résultat **True** ou **False** (tout comme pour la structure alternative). **Elle est évaluée avant chaque exécution du corps de la boucle**. Il importe donc de veiller à la faire évoluer. Il y a lieu de distinguer deux cas particuliers de boucle :

⋆ Si la condition est fausse dès le départ, alors le corps de la boucle ne sera pas du tout exécuté.

⋆ Si la condition reste vraie (parce qu'on la fait mal évoluer ou ne pas évoluer du tout, alors le corps de la boucle sera répété à l'infini.

L'instruction **break** permet de quitter immédiatement la boucle entourante (sortie forcée d'une boucle), ce qui est particulièrement utile pour interrompre des boucles infinies sinon. D'autre part, l'instruction **continue** permet d'interrompre l'exécution au milieu du corps de boucle **sans** quitter la boucle, c'est-à-dire on recommence à évaluer la condition de la boucle.

4.1.2 La boucle for

Lorsqu'on connaît le nombre d'itérations, on peut se servir de la boucle **for**. Elle se sert d'une variable qui prend la valeur de chacun des éléments fournis dans une liste ou un intervalle.

La syntaxe générale de la boucle **for** est la suivante :

```
for <iterator> in <list_of_values>:
    <instruction(s)>
```

À titre d'exemple, on se propose d'afficher les carrés des 10 premiers entiers naturels :

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(i ** 2, end = "␣")       # affichage: 0 1 4 9 16 25 36 49 64 81
```

À chaque itération la variable i prend la valeur de l'élément correspondant dans la liste, puis le corps de la boucle est exécuté. On n'a pas besoin de s'occuper de cette affectation, c'est Python qui s'en charge. Il est maintenant peu commode de devoir écrire tous les éléments de la liste. Pour cela Python permet de générer des listes de façon simple en se servant de la fonction `range`.

Ainsi notre boucle devient plus lisible :

```
for i in range(10):
    print(i ** 2, end = "␣")        # affichage: 0 1 4 9 16 25 36 49 64 81
```

La fonction `range` produit une séquence de nombres **entiers**. La syntaxe complète est :

```
range(start, stop, step)
```

`start` indique la valeur de départ, `stop` indique la valeur d'arrivée (cette valeur ne sera **jamais** dans la liste), `step` indique l'incrément entre deux valeurs successives (elle peut être négative pour générer des listes décroissantes.

L'argument `stop` est obligatoire, alors que `start` (0 par défaut) et `step` (1 par défaut) sont optionnels. Les trois arguments doivent tous être des entiers. Voici quelques exemples d'utilisation :

```
for i in range(10):
    print(i)                    # values 0, 1, ..., 9

for i in range(3, 8):           # values 3, 4, 5, 6, 7
    print(i)

for i in range(2, 17, 3):       # values 2, 5, 8, 11, 14
    print(i)

for i in range(4, 0, -1):       # values 4, 3, 2, 1
    print(i)

for letter in "Hello!":         # values 'H', 'e', 'l', 'l', 'o', '!'
    print(letter)
```

Comme vu plus haut, la boucle **`for`** se charge elle-même des étapes 2, 3 et 4 de la construction d'une boucle (préparation de l'itération suivante, définition du contrôle et initialisation). C'est la raison pour laquelle elle est utilisée de loin plus souvent que la boucle **`while`**.

Remarques :

- ⋆ La taille maximale d'une liste est de $2^{31} - 1$ ou de $2^{63} - 1$ selon l'environnement de programmation.
- ⋆ La liste peut contenir des données de types hétérogènes, p. ex. : `[2, 8, "jeudi", 3.1416]`
- ⋆ Les séquences générées par `range` sont immuables. Pour en faire des vraies listes il faut les convertir par l'instruction `list` (voir chapitre 6.1.4).
- ⋆ Dans le corps de boucle, l'instruction **`break`** permet à tout moment de quitter la boucle, sans passer en revue les éléments restants de la liste.
- ⋆ Dans le corps de boucle, l'instruction **`continue`** permet d'interrompre l'exécution du corps de boucle pour l'élément actuel et de passer tout de suite à l'élément suivant de la liste.

Les compteurs de boucles utilisés sont souvent des variables dont le nom est très court (à une seule lettre, comme p. ex. i). Lorsque le nom du compteur n'est pas utilisé à l'intérieur de la boucle, mais sert simplement à faire répéter plusieurs fois le corps de la boucle, il est de bon usage de choisir comme nom `_` (*underscore*) :

```
for _ in range(5):          # recommended name
    print("Hello!")         # iterator not used in loop
```

4.2 Exercices

Exercice 4.1 Le programme de test de primalité présenté plus haut a l'avantage d'être très simple, mais il est peu efficace (d'ordre $\Theta(n)$, dû au fait que pour un nombre n donné on a besoin de n itérations de boucle, d'après les lignes 11, 12 et 15 du code-source).

Essayer de réduire le nombre d'itérations en affinant le programme en conséquence et de trouver l'ordre minimal pour cet algorithme.

Exercice 4.2 Créer un programme qui affiche x^n sans se servir de l'opérateur $**$. Les nombres $x \in \mathbb{R}$ et $n \in \mathbb{Z}$ sont lus au clavier. Si la puissance en question n'existe pas, un message adéquat est à afficher.

Exercice 4.3 Créer un programme qui calcule et affiche une table de multiplication (*Einmaleins*) de $1 \cdot 1$ jusqu'à $n \cdot n$, où le nombre n est lu au clavier et doit être compris entre 5 et 10. L'affichage doit se faire sous forme de tableau bien formaté, en alignant les nombres à droite.

Exemple : Voici l'affichage pour $n = 6$.

```
   |   1   2   3   4   5   6
---|----------------------
  1|   1   2   3   4   5   6
  2|   2   4   6   8  10  12
  3|   3   6   9  12  15  18
  4|   4   8  12  16  20  24
  5|   5  10  15  20  25  30
  6|   6  12  18  24  30  36
```

Exercice 4.4 Créer un programme qui affiche le nombre de Collatz/Syracuse d'un nombre entier strictement positif lu au clavier.

La suite de Collatz (ou suite de Syracuse) d'un nombre entier $N > 0$ est définie par récurrence de la manière suivante :

$$\begin{cases} u_0 = N \\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases} \end{cases}$$

On répète cette itération jusqu'à tomber sur le nombre 1 et on compte le nombre d'itérations nécessaires.

Exercice 4.5 En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n :

$$n! = n \cdot (n-1) \cdots 2 \cdot 1$$

Par convention, on a $0! = 1$.

Créer un programme qui affiche la factorielle d'un nombre naturel lu au clavier, sans se servir de la fonction `factorial` du module `math`.

Exercice 4.6 Créer un programme qui affiche la somme des chiffres (*Quersumme*) d'un nombre naturel lu au clavier.

Exercice 4.7 Créer un programme qui vérifie si un nombre naturel lu au clavier est un nombre palindrome ou non, sans passer par des chaînes de caractères. Un nombre palindrome garde la même valeur dans les deux sens de lecture Par exemple 121, 88 et 9009 sont des nombres palindromes, mais 69, 110 et 123 n'en sont pas.

Exercice 4.8 Créer un programme qui affiche la somme et la moyenne d'une série de valeurs entières lues au clavier, sachant que la série se termine par un nombre strictement négatif qui lui n'est plus à traiter. Si la moyenne ne peut être calculée, un message adéquat est à afficher.

Exercice 4.9 Créer une programme qui affiche le plus grand commun diviseur (pgcd) de deux nombres naturels non nuls a et b lus au clavier. Pour le calcul, on utilise la méthode d'Euclide (*division euclidienne*). Les formules d'Euclide pour trouver le pgcd de deux nombres naturels par division sont :

$$\begin{cases} \text{pgcd}(a,0) = a \\ \text{pgcd}(a,b) = \text{pgcd}(b,a) \\ \text{pgcd}(a,b) = \text{pgcd}(a \bmod b, b) \text{ si } a > b > 0 \end{cases}$$

Exercice 4.10 Créer un programme qui lit un nombre naturel non nul n et qui calcule et affiche ensuite la décomposition en facteurs premiers de ce nombre.

Exemple : Pour 360, le programme affichera : $360 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5$

Exercice 4.11 Créer un programme qui effectue une conversion décimale $\rightarrow$ binaire. Il lit donc un nombre naturel n en notation décimale, puis il le transforme en notation binaire et l'affiche sous cette forme.

Exemple : Pour 43, la notation binaire affichée sera `101011`, car

$$43 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (101011)_2$$

Exercice 4.12 Créer un programme qui réalise le jeu HiLo :

- ⋆ le programme choisit un nombre secret entre 1 et n, n étant lu au clavier,
- ⋆ à chaque essai de l'utilisateur, le programme affiche le message adéquat : « nombre trop grand », « nombre trop petit », « nombre trouvé après x essais ».

Exercice 4.13 On peut approcher π par la suite de Leibniz :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \cdots$$

Créer un programme qui calcule π avec une précision (nombre de décimales) demandé au clavier et qui affiche π, sa valeur approchée et le nombre de termes nécessaires.

Attention : cette formule n'est pas très efficace : il faut calculer 100 termes pour obtenir un précision de 0,01. Pour 100 décimales, il faudrait plus de termes qu'il n'y a de particules dans l'Univers !

Exercice 4.14 Créer un programme qui dessine chacune des figures ci-dessous, la taille (le nombre de lignes) étant entrée au clavier.

Exemple pour une taille de 5 lignes chacune :

```
a)          b)          c)            d)
*            *          *              *
**           **        ***            ***
***         ***        *****         *****
****       ****       *******         ***
*****     *****      *********         *
```

Exercice 4.15 Créer un programme qui dessine chacune des figures ci-dessous, la taille (le nombre de lignes) étant entrée au clavier.

Exemple pour une taille de 8 lignes chacune :

```
a)          b)          c)          d) (échiquier)
********    ********    ********    X X X X
*      *    **     *    *     **     X X X X
*      *    * *    *    *    * *    X X X X
*      *    *  *   *    *   *  *     X X X X
*      *    *   *  *    *  *   *    X X X X
*      *    *    * *    * *    *     X X X X
*      *    *     **    **     *    X X X X
********    ********    ********     X X X X
```

5 Fonctions

5.1 Introduction

Écrivons un programme qui calcule le nombre de combinaisons de p objets parmi n (avec $0 \leq p \leq n$). Rappelons que ce nombre se détermine par la formule

$$C_n^p = \binom{n}{p} = \frac{n!}{p! \cdot (n-p)!}$$

Nous constatons que pour chaque calcul d'un nombre de combinaisons nous devons effectuer trois fois le calcul d'une factorielle et ainsi répéter trois fois le même code. Les fonctions servent à éviter ces redondances dans les programmes. Grâce à elles, il suffit de noter une seule fois le code de calcul de la factorielle et de l'utiliser à chaque besoin. Établissons d'abord (sans recourir à une fonction prédéfinie dans le module mathématique de Python) le calcul de la factorielle $n!$ sous forme de fonction.

```
def fact(n):
    x = 1
    for i in range(2, n + 1):
        x *= i
    return x
```

L'identificateur n est un paramètre dont la valeur est fixée lors de chaque appel de la fonction. La fonction `fact` retourne une valeur qui sera transmise au programme appelant. Une fonction, une fois définie comme telle, peut être utilisée comme suit :

```
def comb(n, p):
    if 0 <= p <= n:
        return fact(n) // (fact(p) * fact(n - p))
    else:
        return "mauvais arguments"
```

Ainsi, par exemple le nombre de tirages possibles du *Samstagslotto* allemand (*6 aus 49*) est égal à

```
>>> print(comb(49, 6))
 13983816
```

Lorsque les arguments sont mal choisis, un message est renvoyé :

```
>>> print(comb(4, 5))
 mauvais arguments
```

5.2 Définition

Une fonction est une partie de code d'un programme que nous pouvons utiliser (appeler) plusieurs fois. Chaque fois que nous l'utilisons, nous pouvons lui transmettre d'autres valeurs comme arguments qu'elle utilisera durant son exécution. Le code de la fonction est placé plus haut dans le code-source que l'appel effectif de la fonction (c'est-à-dire au moment de l'appel de la fonction par l'interpréteur du programme sa définition doit déjà avoir été lue).

```
def <name_of_function>(param_1, ..., param_n):
    <instruction(s)>
```

Normalement, le corps d'une fonction contient une ou plusieurs occurrences de l'instruction **return**, qui permet de renvoyer la réponse déterminée par la fonction. Après l'instruction **return**, l'exécution de la fonction est terminée ; il est donc possible de quitter une fonction à différents endroits du corps.

```
def <name_of_function>(param_1, ..., param_n):
    <instruction(s)>
    return <answer>
```

Lorsqu'une fonction ne contient aucune instruction **return**, la valeur implicite **None** est renvoyée. Dans d'autres langages de programmation (mais pas dans Python), de telles fonctions sont parfois appelées *procédures*.

Une fonction qui ne reçoit aucun argument garde néanmoins ses parenthèses, qui indiquent qu'il s'agit bien d'une fonction.

```
def <name_of_function>():
    <instruction(s)>
    return <answer>
```

Pour appeler une fonction, on écrit par exemple :

```
my_result = <name_of_function>(arg_1,..., arg_n)
```

Il importe de distinguer les deux notations suivantes et la terminologie correspondante :

- ⋆ `f(x)` : la **fonction** f est appelée avec un argument x ;
- ⋆ `f(x, y)` : la **fonction** f est appelée avec les arguments x et y ;
- ⋆ `x.f()` : la **méthode** f de l'objet x est appelée ;
- ⋆ `x.f(y)` : la **méthode** f de l'objet x est appelée, compte tenu de l'argument y.

Les méthodes seront traitées plus tard dans le cours.

5.3 Exercices

Exercice 5.1 Écrire une fonction `max3` qui renvoie le maximum de trois nombres fournis comme arguments. Faire les tests explicitement dans le corps de la fonction, sans recourir à une fonction prédéfinie comme `max`.

Exercice 5.2 Écrire une fonction `pgcd` qui reçoit comme arguments deux nombres naturels non nuls et qui calcule et renvoie leur *plus grand commun diviseur*.

Exercice 5.3 Écrire une fonction `pgcd3` qui reçoit come arguments trois nombres entiers et qui calcule et renvoie le *plus grand commun diviseur* de ces trois nombres, respectivement un message d'erreur lorsque les nombres entrés ne sont pas tous strictement positifs. On pourra utiliser la fonction `pgcd` de l'exercice précédent, ainsi que la propriété

$$\mathrm{pgcd}(a, b, c) = \mathrm{pgcd}(\mathrm{pgcd}(a, b), c)\,.$$

Exercice 5.4 Écrire une fonction `sum_of_integers` qui reçoit comme arguments deux nombres a et b quelconques (pas forcément entiers) et qui calcule et renvoie la somme de tous les entiers compris entre a et b, bornes comprises. Le programme doit traiter correctement les différents cas $a < b$, $a = b$, $a > b$.

Exercice 5.5 * Écrire une fonction `geo3` qui reçoit comme arguments trois nombres quelconques x, y, z et qui calcule et renvoie la moyenne géométrique $G(x, y, z)$ de ces trois nombres. On définit :

$$G(x,y,z) = \begin{cases} \sqrt[3]{xyz} & \text{si } x,y,z \text{ sont non nuls et ont tous le même signe,} \\ 0 & \text{sinon} \end{cases}$$

Exercice 5.6 * Écrire une fonction `add_one_sec` qui reçoit comme arguments trois nombres naturels h, m, s représentant une heure correcte ($0 \leq h < 24$, $0 \leq m < 60$ et $0 \leq s < 60$) et qui renvoie un string dans le format « `hh mm ss` » représentant l'heure qu'il sera une seconde plus tard.

- ⋆ si $h = 3$, $m = 14$ et $s = 15$, la réponse sera « `03 14 16` » ;
- ⋆ si $h = 11$, $m = 13$ et $s = 59$, la réponse sera « `11 14 00` » ;
- ⋆ si $h = 12$, $m = 59$ et $s = 59$, la réponse sera « `13 00 00` » ;
- ⋆ si $h = 23$, $m = 59$ et $s = 59$, la réponse sera « `00 00 00` » ;
- ⋆ si $h = 13$, $m = 40$ et $s = 60$, la réponse sera « `Arguments invalides !` » ;

Exercice 5.7 ** Écrire un programme qui pour une fonction f intégrée au code-source, demande à l'utilisateur d'entrer une valeur initiale x_0 et qui essaie d'approcher une racine de la fonction f par l'*algorithme de Newton* : pour $i = 0, 1, 2, \ldots$

- ⋆ si $|f(x_i)| < \varepsilon$ pour un petit nombre $\varepsilon > 0$, on admet que x_i est une bonne approximation d'une racine de f et on arrête les calculs ;
- ⋆ sinon, on calcule $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ et on espère que la suite $(x_i)_i$ converge vers une racine de f.

Pour calculer le nombre-dérivé $f'(x_i)$, on peut soit implémenter séparément la fonction-dérivée de f dans le code-source, soit approcher ce nombre par le calcul

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

pour un petit nombre $h > 0$.

5.4 Paramètres optionnels

Rappelons que certaines expressions acceptent un nombre variable d'arguments. Ainsi on peut écrire aussi bien `range(2, 9, 1)` que `range(2, 9)` pour décrire l'intervalle des nombres entiers de 2 à 8. Dans l'expression `range(2, 9)`, le troisième argument n'est pas fourni et on en déduit qu'il vaut 1 par défaut.

Les fonctions que nous définissons nous-mêmes peuvent également accepter des arguments par défaut. La fonction suivante calcule la longueur d'une diagonale d'un rectangle (en deux dimensions) ou d'un parallélépipède rectangulaire (en trois dimensions) en appliquant la formule de Pythagore :

```
def pyth(a, b, c = 0):
    return (a ** 2 + b ** 2 + c ** 2) ** (1 / 2)
```

Remarque : au lieu d'élever à la puissance 1/2 pour calculer la racine carrée, on aurait également pu importer la fonction `sqrt` de la bibliothèque `math` :

```
from math import sqrt
def pyth(a, b, c = 0):
    return sqrt(a ** 2 + b ** 2 + c ** 2)
```

Lors de l'appel de la fonction `pyth`, on peut fournir soit deux, soit trois arguments. Lorsqu'on ne donne que deux arguments, la valeur par défaut (qui vaut 0 d'après la définition de la fonction) est utilisée pour le paramètre c.

```
>>> print(pyth(5, 12))
 13.0
>>> print(pyth(4, 7, 4))
 9.0
```

En général, on peut avoir un ou plusieurs paramètres avec valeurs par défaut. Ils suivent ceux dont les arguments sont obligatoires lors de l'appel de la fonction.

```
def <name of function>(param_1, ..., param_m, param_n = 42):
    <instruction(s)>
    return <answer>
```

5.5 Nombre indéterminé de paramètres *

Rappelons que certaines expressions acceptent un nombre variable d'arguments. Ainsi on peut appeler `print(...)` avec un nombre quelconque d'arguments qui seront tous affichés, l'un après l'autre et séparés entre eux par une espace.

Les fonctions que nous définissons nous-mêmes peuvent également accepter un nombre quelconque d'arguments. La fonction suivante accepte comme premier argument une base $b \geq 2$ et comme arguments supplémentaires des chiffres représentant un nombre entier positif en base b. Chaque chiffre doit être un entier compris entre 0 et $b-1$. La fonction détermine le nombre ainsi représenté et le renvoie, de sorte qu'il puisse être affiché en notation décimale.

```
def nombre_base(b, *c):
    x = 0
    for i in c:
        if 0 <= i < b:
            x = x * b + i
        else:
            return "mauvais arguments"
    return x
```

Ainsi, `nombre_base(10, 3, 4, 5, 6)` renvoie 3456, et `nombre_base(7, 1, 2, 3)` renvoie 66, car $1 \cdot 7^2 + 2 \cdot 7 + 3 = 66$.

L'astérisque devant le paramètre c indique qu'il représente des arguments en nombre quelconque. On accède au contenu du paramètre c en parcourant son contenu dans une boucle.

5.6 Variables locales et globales *

Une fonction reçoit le plus souvent des arguments qui sont affectés aux paramètres qui jouent le rôle de variables à l'intérieur de la fonction. Le passage argument ⇒ paramètre **est toujours réalisé comme une affectation de variables** et suit les mêmes règles.

Lorsque les paramètres ou des variables définies par la fonction portent le même nom que les arguments ou d'autres variables extérieures à la fonction, ils cachent leurs homonymes extérieurs et sont ainsi appelés « variables locales ».

Quelques exemples pour illustrer ce concept extrêmement important :

```
def f(x):
    x *= 2
    y = 20
    return x

x = 1
y = 10
print(f(x))
print(x, y)
```

Ce programme affiche

```
2
1 10
```

En effet, le paramètre x de la fonction f est local à f et ne modifie pas le contenu de la variable x à l'extérieur de la fonction. De même, la variable y définie par la fonction est locale à f et n'a rien à faire avec la variable extérieure y, dont le contenu n'est pas accessible à f et ne peut donc pas être modifié par mégarde.

```
def f(x):
    global y
    x *= 2
    y = 20
    return x

x = 1
y = 10
print(f(x))
print(x, y)
```

Ce programme affiche

```
2
1 20
```

Maintenant, y est déclaré comme variable globale, c'est-à-dire la même variable y est utilisée à l'intérieur comme à l'extérieur de la fonction f. Le contenu de y est ainsi modifié par f.

Cette technique ne s'applique pas aux paramètres : on ne peut pas accéder à l'original (la variable extérieure x) via le paramètre x en écrivant `global x` !

6 Structures de données élémentaires

6.1 Listes

6.1.1 Problème introductif

On veut écrire un programme qui permet d'effectuer certaines opérations (somme, différence, produit, évaluation pour une valeur x donnée, ...) sur des polynômes

$$P(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0 \,.$$

Les types de variables utilisés jusqu'à maintenant ne sont pas suffisants pour pouvoir stocker de façon élégante les coefficients d'un polynôme, d'autant plus que leur nombre n'est pas fixe, mais dépend du degré n du polynôme.

Nous allons utiliser une *liste* (type `list` en Python) pour stocker les différents coefficients a_i. Les coefficients sont stockés de telle manière à ce que l'indice du coefficient a_i dans la liste corresponde à son propre indice i. Cela est réalisé trivialement, car les indices d'éléments dans une liste sont toujours numérotés à partir de la valeur initiale 0.

Dans un premier temps, nous allons nous contenter d'écrire un programme qui demande à l'utilisateur d'entrer le degré d'un polynôme $P(x)$ et ensuite les différents coefficients.

Solution :

```
d = int(input("Entrez le degré du polynôme : "))
poly = []
for i in range(d + 1):
    poly.append(float(input(f"Entrez le coefficient d'indice {i} : ")))
```

6.1.2 Création et utilisation des listes

⋆ On utilise les crochets comme délimiteurs d'une liste. Ainsi, pour créer une liste vide :

```
my_list = []
```

⋆ Voici une liste qui contient deux strings :

```
my_list = ["Hello", "world"]
```

⋆ On peut rajouter un élément à la fin d'une liste donnée, en utilisant la méthode `append` :

```
my_list.append("again")
```

⋆ Lorsqu'on affiche directement une liste, une conversion implicite en string est effectuée :

```
>>> print(my_list)
 ['Hello', 'world', 'again']
```

⋆ On peut accéder à un élément d'une liste via son indice (à écrire entre crochets). Les indices sont numérotés à partir de 0.

```
>>> print(my_list[2])
 again
```

⋆ Une liste est dite *mutable* (modifiable), c'est-à-dire on peut modifier un élément sans toucher au reste de la liste :

```
my_list[0] = "Goodbye"
```

⋆ La méthode `count` compte le nombre d'occurrences d'un élément dans une liste :

```
>>> print(my_list.count("Goodbye"))
 1
```

⋆ L'opérateur **`in`** vérifie si un élément appartient à une liste :

```
>>> print("Hello" in my_list)
 False
```

⋆ La méthode `index` renvoie l'indice de la première occurrence d'un élément dans une liste et produit une erreur (exception `ValueError`) si l'élément n'y existe pas.

```
>>> print(my_list.index("again"))
 2
```

⋆ La fonction `len` renvoie le nombre d'éléments d'une liste :

```
>>> print(len(my_list))
 3
```

⋆ On peut enlever un élément de la liste en précisant son indice. Les éléments à droite de l'élément enlevé sont automatiquement décalés d'une position vers la gauche.

```
>>> del my_list[1]
>>> print(my_list)
 ['Goodbye', 'again']
```

⋆ On peut également enlever la première occurrence d'un élément décrit explicitement. Lorsque l'élément n'existe pas dans la liste, une erreur (exception `ValueError`) est produite.

```
my_list.remove("again")
```

⋆ La méthode `reverse` inverse l'ordre des éléments dans une liste :

```
>>> my_list = [1, 7, 5, 9, 3]
>>> my_list.reverse()
>>> print(my_list)
 [3, 9, 5, 7, 1]
```

⋆ Pour trier une liste *in situ*, on utilise la méthode `sort` :

```
>>> my_list.sort()
>>> print(my_list)
 [1, 3, 5, 7, 9]
```

⋆ L'opérateur + sert à combiner (concaténer) plusieurs listes en une seule :

```
>>> my_new_list = my_list + [0, 2]
>>> print(my_new_list)
 [1, 3, 5, 7, 9, 0, 2]
```

On peut également utiliser la méthode `extend` pour attacher une liste à une autre liste :

```
>>> my_new_list.extend([4, 6])
>>> print(my_new_list)
 [1, 3, 5, 7, 9, 0, 2, 4, 6]
```

⋆ La méthode `pop` enlève (et renvoie) le dernier élément d'une liste :

```
>>> element = my_list.pop()
>>> print(element)
>>> print(my_list)
 9
 [1, 3, 5, 7]
```

⋆ La méthode `insert` insère un élément à une position donnée (avec décalage d'éléments vers la droite, si nécessaire) :

```
>>> my_list.insert(3, "Hello")
>>> print(my_list)
 [1, 3, 5, 'Hello', 7]
```

⋆ On peut parcourir tous les éléments d'une liste de la gauche vers la droite dans une boucle `for` :

```
for element in my_list:
    ...
```

Si l'on veut parcourir la liste dans le sens inverse, il faut inverser l'ordre avec la fonction `reversed` (à ne pas confondre avec la méthode `reverse` vue plus haut) :

```
for element in reversed(my_list):
    ...
```

⋆ Par analogie à la méthode `sort`, la fonction `sorted` génère une nouvelle liste triée sans modifier la liste donnée.

⋆ Il existe également des fonctions mathématiques applicables aux listes : Par exemple, `min(my_list)` calcule le minimum des éléments (supposés numériques) d'une liste, `max(my_list)` calcule le maximum et `sum(my_list)` calcule la somme des éléments.

6.1.3 Exercices simples sur les listes

Exercice 6.1 Écrire un programme qui demande à l'utilisateur d'entrer 5 réels, les stocke dans une liste et affiche à la fin le contenu de la liste.

Exercice 6.2 Remplir une liste avec 20 entiers aléatoires compris entre 1 et 100. Afficher la liste ; puis calculer le minimum, le maximum, la somme et la moyenne, et afficher ces nombres.

Exercice 6.3 Remplir une liste avec 30 entiers aléatoires compris entre 10 et 25. Déterminer le nombre d'occurrences (fréquence) d'un entier entré par l'utilisateur. Inverser ensuite l'ordre des nombres dans la liste et afficher la liste réarrangée.

Exercice 6.4 * Écrire un programme qui demande à l'utilisateur d'entrer une année et le numéro d'un jour dans cette année. Le programme affichera le jour et le mois correspondant, tout en tenant compte des années bissextiles le cas échéant.

Exercice 6.5 – Affichage d'un polynôme : Adapter le programme introductif afin qu'il affiche le polynôme sous forme de texte.

Exemple : La liste [1, 2, 3] sera affichée en tant que polynôme comme `3 x^2 + 2 x + 1`.

* Optimiser le programme pour que le polynôme soit affiché de manière optimale (sans signes + ou − juxtaposés, sans coefficient 1 superflu et sans termes nuls superflus).

Exercice 6.6 – Évaluation d'un polynôme : Écrire une fonction qui reçoit comme arguments un polynôme et une valeur réelle, et qui évalue le polynôme en cette valeur en utilisant le schéma de Horner.

Exercice 6.7 – Simulation d'un tirage de *Samstagslotto* : Écrire un programme qui demande à l'utilisateur d'entrer six nombres entiers distincts entre 1 et 49. Les nombres sont stockés dans une liste. Le programm vérifiera que les nombres sont distincts deux à deux.

Le programme effectue ensuite le tirage au hasard de six nombres (entre 1 et 49), tout en veillant à ce qu'il y ait effectivement six nombres différents. Une nouvelle liste est utilisée pour mémoriser les nombres du tirage.

Le programme affiche les deux listes et indique finalement combien de nombres le joueur a deviné correctement.

6.1.4 Création automatisée de listes

Les intervalles de valeurs utilisés pour les boucles permettent de créer des listes de façon automatisée. Il suffit de convertir un `range` explicitement en `list` pour obtenir la liste correspondante :

```
my_list = list(range(6))                # [0, 1, 2, 3, 4, 5]
my_list = list(range(8, -4, -3))        # [8, 5, 2, -1]
```

On rappelle que la valeur du 2e argument ne figure plus dans la liste (principe d'intervalle semi-ouvert $[a, b[$).

6.1.5 Indices négatifs et sous-listes

Nous avons vu plus haut qu'un indice $i \geq 0$ accède à l'élément numéro $i + 1$ d'une liste :

```
>>> my_list = [2, 3, 5, 7, 11, 13]
>>> print(my_list[4])
 11
```

Des indices strictement négatifs accèdent aux éléments à partir de la fin ; -1 renvoie au dernier élément, -2 à l'avant-dernier, etc. :

```
>>> print(my_list[-1], my_list[-3])
 13 7
```

Il est également possible d'extraire des sous-listes. On utilise comme indice un intervalle semi-ouvert (qu'on noterait en mathématiques $[a, b[$) :

```
>>> print(my_list[1:4])
 [3, 5, 7]
```

La sous-liste précédente contient les éléments d'indice 1 à 3, c'est-à-dire les 2e, 3e et 4e éléments.

```
>>> print(my_list[2:-1])
 [5, 7, 11]
```

La sous-liste précédente contient tous les éléments à partir du 3e (indice 2) jusqu'à l'avant-dernier (le dernier, d'indice -1, étant exclu).

On peut omettre l'indice de départ ou l'indice d'arrivée, si l'on veut construire des sous-intervalles à partir du début ou jusqu'à la fin de la liste :

```
>>> print(my_list[:4])
 [2, 3, 5, 7]
```

Voici quelques exemples supplémentaires :

```
my_list = ["a", "b", "c", "d", "e"]
print(my_list[1:3])       # ['b', 'c']
print(my_list[:2])        # ['a', 'b']
print(my_list[3:])        # ['d', 'e']
print(my_list[2:-1])      # ['c', 'd']
```

6.1.6 Copier une liste

On récupère une liste entière en omettant les deux extrémités de l'intervalle d'indices :

```
>>> print(my_list[:])
 [2, 3, 5, 7, 11, 13]
```

Cette notation paraît triviale ou même superflue à première vue, mais elle est essentielle si l'on veut copier des listes. En effet, en écrivant

```
my_list = [2, 3, 5, 7, 11, 13]
copy_list = my_list
```

la copie renvoie à la **même** liste que la variable de départ ! Ainsi, en écrivant

```
copy_list[0] = 1
```

on a également modifié la liste de départ :

```
>>> print(my_list)
 [1, 3, 5, 7, 11, 13]
```

La variable `copy_list` n'est qu'un alias de la variable `my_list`, ou autrement dit, les deux variables `my_list` et `copy_list` pointent vers le **même** objet dans la mémoire. Un compteur interne comptabilise combien de fois un tel objet de la mémoire est référencé. Ainsi, lorsqu'on détruit une variable pointant vers un objet, ce dernier n'est pas effacé de la mémoire aussi longtemps que d'autres variables y pointent encore. Après l'instruction

```
del copy_list
```

la variable `my_list` (et surtout son contenu) existe encore :

```
>>> print(my_list)
 [1, 3, 5, 7, 11, 13]
```

Par contre, lorsqu'on veut vraiment copier une liste, et créer ainsi une nouvelle variable indépendante de la première, il faut écrire :

```
my_list = [2, 3, 5, 7, 11, 13]
copy_list = my_list[:]
```

Maintenant on peut modifier la copie sans toucher à l'original :

```
>>> copy_list[0] = 1
>>> print(my_list)
 [2, 3, 5, 7, 11, 13]
```

Il importe de savoir qu'il s'agit d'une **copie creuse** (*shallow copy*) de la liste originale, c'est-à-dire les éléments de la liste initiale ne sont copiés que lorsqu'ils ne sont pas *mutables*. Lorsque les éléments de la liste initiale sont eux-mêmes des listes, des dictionnaires etc. (c'est-à-dire des objets *mutables*), la liste copiée contiendra des références vers les objets originaux et non pas des copies véritables. Ce sujet sera approfondi dans le chapitre suivant (copie de matrices).

6.1.7 Modification de (sous-)listes

Nous avons vu plus haut que les éléments d'une liste sont modifiables, l'accès se faisant par l'indice :

```
>>> my_list = [1, 3, 5, 7, 9]
>>> my_list[2] = 11
>>> print(my_list)
 [1, 3, 11, 7, 9]
```

On peut aussi remplacer une sous-liste par un contenu nouveau :

```
>>> my_list = [1, 3, 5, 7, 9]
>>> my_list[2:4] = [11, 12, 13]
>>> print(my_list)
 [1, 3, 11, 12, 13, 9]
```

On vient de remplacer la sous-liste $[5, 7]$ par $[11, 12, 13]$. La taille d'une liste peut donc changer lors d'une telle affectation.

Attention ! Lorsqu'on veut remplacer un seul élément par plusieurs éléments dans une liste, il faut bien distinguer les deux cas suivants :

```
>>> my_list = [1, 3, 5, 7, 9]
>>> my_list[2] = [11, 12, 13]
>>> print(my_list)
 [1, 3, [11, 12, 13], 7, 9]
```

On vient de remplacer l'élément 5 par $[11, 12, 13]$ et on a ainsi créé, peut-être sans le vouloir, une liste imbriquée dans la liste de départ, qui, elle, contient encore 5 éléments, la liste imbriquée comptant comme un seul élément de la liste enveloppante. D'autre part,

```
>>> my_list = [1, 3, 5, 7, 9]
>>> my_list[2:3] = [11, 12, 13]
>>> print(my_list)
 [1, 3, 11, 12, 13, 7, 9]
```

donne une liste où l'élément 5 (ciblé par `[2:3]`) a été remplacé par plusieurs éléments nouveaux, ce qui modifie la taille de la liste initiale.

6.1.8 Utilisation de listes comme arguments dans des fonctions

Nous avons vu plus haut que le passage argument ⇒ paramètre **est toujours réalisé comme une affectation de variables**. Pour les listes, qui sont *mutables* et non copiées lors d'une affectation normale, les conséquences sont importantes :

```
def f(li):
    li += [0]
    li[0] = 7
    return len(li)

my_list = [1, 2, 3, 4]
print(f(my_list))
print(my_list)
```

Ce programme affiche

```
5
[7, 2, 3, 4, 0]
```

Le paramètre `li` contient seulement une référence vers la variable `my_list`, donc la liste originale est modifiée par la fonction f. L'opérateur `+=` est équivalent à la méthode `extend`, et si la liste à attacher ne contient qu'un seul élément, on aurait aussi pu utiliser la méthode `append` : à la deuxième ligne du code, on aurait pu écrire tout aussi bien `li.extend([0])` ou `li.append(0)`, avec les mêmes effets sur `my_list`.

```
def f(li):
    li = li + [0]
    li[0] = 7
    return len(li)

my_list = [1, 2, 3, 4]
print(f(my_list))
print(my_list)
```

Ce programme affiche

```
5
[1, 2, 3, 4]
```

À la deuxième ligne du code, une **nouvelle** liste est générée par `li + [0]`, et cette nouvelle liste est stockée dans la variable **locale** dont le nom est `li` (membre de gauche de l'affectation) et qui coïncide « par hasard » avec le nom du paramètre. Comme il s'agit d'une nouvelle liste et que la variable `li` à laquelle elle est affectée est **locale** par construction, `li` est maintenant détachée de la variable extérieure `my_list` vers laquelle `li` référençait jusqu'à présent. Les instructions suivantes ne modifient donc plus la variable `my_list`.

6.1.9 Exercices récapitulatifs sur les listes

Exercice 6.8 Écrire un programme qui lit deux nombres naturels non nuls a et n et qui génère la liste des n premiers nombres premiers $\geq a$.

Exercice 6.9 Écrire un programme qui lit un entier $n \geq 2$ et ensuite lit (ou génère de manière automatisée à l'aide de valeurs aléatoires) une liste à n nombres réels strictement positifs et inférieurs à 100. Le programme calculera et affichera alors les moyennes suivantes de ces nombres : la moyenne arithmétique, la moyenne géométrique, la moyenne harmonique et la moyenne quadratique.

Rappel : `https://fr.wikipedia.org/wiki/Moyenne#Les_diff.C3.A9rentes_moyennes`

Exercice 6.10 * Écrire un programme qui lit un entier $n \geq 2$ et ensuite lit (ou génère de manière automatisée à l'aide de valeurs aléatoires) une liste à n nombres réels. Le programme calculera et affichera alors la médiane, le premier quartile et le troisième quartile de cette liste de nombres.

6.2 Strings

Un *string* est une chaîne de caractères. Par caractère, on entend tout symbole pouvant être représenté dans une langue quelconque, y compris les lettres (majuscules ou minuscules, standard ou accentuées), les chiffres, etc.

Attention ! Une espace compte également comme un caractère.

6.2.1 Notions de base

Les chaînes de caractères, de type `str` en Python, ont comme délimiteurs des apostrophes `'...'` ou des guillemets `"..."`. Lorsque le caractère choisi comme délimiteur apparaît lui-même dans la chaîne, il faut le faire précéder d'un backslash `\` :

```
my_string = 'aujourd\'hui'
```

Rappelons brièvement comment créer un string par affectation :

```
my_string_1 = 'hello'
my_string_2 = "world"
```

Le caractère spécial `\n` (*newline*) représente le passage à la ligne suivante.

Une chaîne peut s'étendre sur plusieurs lignes, même dans le code-source, sans que l'on doive forcément recourir à `\n`. Cette chaîne est alors délimitée par des apostrophes ou des guillemets **triples**. D'autre part, on peut concaténer (joindre) des chaînes avec l'opérateur d'addition $+$ et répéter des extraits plusieurs fois avec l'opérateur de multiplication $*$:

```
my_string_3 = """first␣row␣of␣this
string␣that␣spans␣over
three␣rows"""
my_string_4 = "mu" + 5 * "ha" + 2 * "!"
```

Le dernier exemple crée le string `"muhahahahaha!!"`.

Des chaînes de caractères littérales écrites qui se suivent directement sont concaténées implicitement :

```
my_string_5 = "aujourd'hui" "␣et␣demain"
my_string_6 = "aujourd'hui" + "␣et␣demain"
```

Ces deux strings contiennent chacun le texte `"aujourd'hui et demain"`.

6.2.2 Utilisation simple des strings

En Python, un string est une liste de caractères qui n'est pas *mutable* (non modifiable). On peut accéder aux différents caractères via leur indice respectif (compté à partir de 0, comme pour les listes), mais on ne peut pas modifier un caractère dans un string donné. Pour modifier une chaîne il faut donc construire une nouvelle chaîne qui contient les changements désirés.

```
my_string_1 = "Hello␣world"
my_string_2 = my_string_1
```

On peut considérer le deuxième string comme étant une copie du premier. Lorsqu'on remplace le contenu d'un de ces deux strings par une réaffectation, le contenu de l'autre reste inchangé.

Exemples d'utilisation de strings :

```
print(my_string_1[4])            # 'o'
print(my_string_1[-1])           # 'd'
new_string = my_string_1[2:6]    # 'llo '
new_string = my_string_1[4:-2]   # 'o wor'
new_string = my_string_1[:3]     # 'Hel'
new_string = my_string_1[2:]     # 'llo world'
```

Les fonctions/méthodes vues pour les listes sont encore valables pour les strings, sous condition qu'elles ne modifient pas des éléments (caractères) du string. On peut par ailleurs convertir des variables quelconques en string à l'aide de la fonction `str`.

```
n = len(my_string_1)             # 11
m = my_string_1.index("␣")       # 5
new_string = str(n) + str(m)     # 115
```

Précisons encore que Python ne connaît pas de type « caractère » (comme `char` en C ou `Character` en Swift), mais un caractère isolé est tout simplement assimilé à une chaîne de longueur 1. Ainsi, le caractère `s[0]` et l'extrait `s[0:1]` de la chaîne non vide s représentent exactement la même chose, à savoir un string à 1 caractère.

On peut cependant parcourir les caractères d'un string dans une boucle **`for`**, tout comme on peut parcourir les éléments d'une liste :

```
for character in my_string_1:
    ...
for character in reversed(my_string_1):    # reversed order
    ...
```

Dans ces boucles, la variable `character` contient toujours un string à un caractère.

6.2.3 Méthodes utiles

Il existe de nombreuses méthodes applicables aux strings. Voici les plus importantes :

⋆ `s.capitalize()` renvoie une copie de s dont le premier caractère est transformé en majuscule et toutes les autres lettres de s en minuscules.

`s.lower()` renvoie une copie de s dont toutes les lettres sont transformées en lettres minuscules.

`s.upper()` renvoie une copie de s dont toutes les lettres sont transformées en lettres majuscules.

```
s1 = "my␣TINY␣text."
s2 = s1.capitalize()        # 'My tiny text.'
s2 = s1.lower()             # 'my tiny text.'
s2 = s1.upper()             # 'MY TINY TEXT.'
```

⋆ `s.strip()` renvoie une copie de s, où les caractères invisibles (*whitespaces*, p. ex. les espaces) sont enlevés au début et à la fin.

`s.strip(t)` renvoie une copie de s, où les caractères énumérés dans la chaîne t sont enlevés au début et à la fin (de la chaîne s).

```
s2 = "␣␣text␣␣␣".strip()    # 'text'
s2 = s1.strip("met.")       # 'y TINY tex'
```

⋆ `s.split(t)` renvoie une liste de sous-chaînes obtenue à partir de s ; la chaîne est découpée aux endroits où la sous-chaîne t apparaît ; si t apparaît plusieurs fois consécutivement, des sous-chaînes vides sont générées à l'emplacement correspondant dans la liste renvoyée.

`s.split()` renvoie une liste de mots obtenue à partir de s, c'est-à-dire s est découpé aux endroits contenant des caractères invisibles (*whitespaces*) ; **contrairement** à ce qui se passe pour `s.split(t)`, des caractères invisibles juxtaposés ne génèrent pas de mots vides dans la liste.

```
li = s1.split()             # ['my', 'TINY', 'text.']
li = s1.split("t")          # ['my TINY ', 'ex', '.']
```

⋆ `s.find(t)` renvoie l'indice de la première occurrence de la chaîne t dans la chaîne s. Si t n'est pas trouvé dans s, la valeur -1 (et pas d'erreur) est renvoyée.

`s.find(t, a, b)` limite la recherche de t à la sous-chaîne `s[a:b]`.

```
n = s1.find("TI")           # 3
n = s1.find("t", 9, -1)     # 11
```

Pour tester simplement si t est contenu dans s, sans vouloir connaître la position précise, on préférera la notation compacte

```
t in s
```

qui renvoie directement **`True`** resp. **`False`**, plutôt que d'écrire

```
s.find(t) >= 0
```

⋆ `s.replace(t, u)` renvoie une copie de s où toutes les occurrences de la sous-chaîne t sont remplacées par la sous-chaîne u.

⋆ `s.replace(t, u, n)` renvoie une copie de s où les n premières occurrences de la sous-chaîne t sont remplacées par la sous-chaîne u.

```
s2 = "abababa".replace("ab", "c")     # 'ccca'
s2 = "abababa".replace("a", "c", 2)   # 'cbcbaba'
```

⋆ `s.isalpha()` renvoie **`True`** si s contient au moins un caractère et que tous ses caractères sont des lettres alphabétiques ; la méthode renvoie **`False`** sinon.

`s.isdigit()` renvoie **`True`** si s contient au moins un caractère et que tous ses caractères sont des chiffres ; la méthode renvoie **`False`** sinon.

`s.isalnum()` renvoie **`True`** si s contient au moins un caractère et que tous ses caractères sont des lettres alphabétiques ou des chiffres ; la méthode renvoie **`False`** sinon.

`s.islower()` renvoie **`True`** si s contient au moins une lettre et que toutes ses lettres sont des lettres minuscules ; la méthode renvoie **`False`** sinon.

`s.isupper()` renvoie **`True`** si s contient au moins une lettre et que toutes ses lettres sont des lettres majuscules ; la méthode renvoie **`False`** sinon.

`s.isspace()` renvoie **`True`** si s contient au moins un caractère et que tous ses caractères sont invisibles (*whitespaces*) ; la méthode renvoie **`False`** sinon.

6.2.4 Exercices

Exercice 6.11 Écrire une fonction `longest_string` qui retourne la chaîne de caractères la plus longue de trois chaînes fournies.

Exercice 6.12 Écrire une fonction `no_vowels` qui détecte si une chaîne de caractères ne contient aucune voyelle. (Les voyelles sont les lettres a, e, i, o, u, A, E, I, O, U).

Exercice 6.13 Écrire une fonction qui compte le nombre de *chiffres* (caractères $0, 1, \ldots, 9$) dans une chaîne de caractères.

Écrire deux variantes de cette fonction : La première compte simplement le total de chiffres trouvés dans la chaîne. La deuxième fonction renvoie une **liste** de 10 valeurs indiquant le nombre de chiffres 0 trouvés, le nombre de chiffres 1 trouvés, etc.

6.3 Exercices supplémentaires

Exercice 6.14 * Écrire une fonction moyenne qui calcule la moyenne des notes de devoirs en classe. La fonction accepte comme seul argument une liste de nombres (on suppose que tous les nombres sont des entiers entre 1 et 60) et elle renvoie un entier, à savoir la moyenne arrondie (vers le haut) des notes fournies.

Exercice 6.15 – Opérations sur des polynômes *

Écrire un programme qui demande à l'utilisateur d'entrer deux polynômes et les stocke dans deux listes, cf. début du chapitre.

Le programme calculera ensuite la somme, la différence et le produit de ces deux polynômes et affichera les polynômes obtenus. On veillera à afficher les polynômes de manière optimale (sans signes + ou − juxtaposés, sans coefficient 1 superflu et sans termes nuls superflus).

7 Structures de données avancées

7.1 Tuplets

7.1.1 Notation

Nous avons rencontré dans le chapitre précédent deux types de données *composés* : les chaînes de caractères (string, noté `str` en Python), qui sont composées de caractères, et les listes (`list`), qui sont composées d'éléments de n'importe quel type. Alors qu'il n'est pas possible de changer les caractères à l'intérieur d'une chaîne existante, on peut modifier directement les éléments à l'intérieur d'une liste. En d'autres termes, les listes sont dites *mutables*, alors que les chaînes de caractères sont *non-mutables*.

Python propose un type de données appelé *tuplet* (en anglais : *tuple*), qui est semblable à une liste mais qui, comme les chaînes, n'est pas modifiable.

Syntaxiquement, un tuplet est une collection d'éléments séparés par des virgules et délimités par des parenthèses ordinaires :

```
>>> t = (10, 20, 30)
>>> print(t)
 (10, 20, 30)
```

Le plus souvent, lorsqu'aucune ambiguïté n'est possible, on peut laisser de côté les parenthèses lors des affectations :

```
>>> u = 11, 21, 31
>>> u
 (11, 21, 31)
```

L'interpréteur a correctement compris que les trois nombres séparés par des virgules forment implicitement un tuplet. Mais même si les parenthèses sont souvent facultatives, il est recommandé de les écrire lorsqu'elles aident à augmenter la lisibilité du code-source.

L'extraction des éléments d'un tuplet se fait aussi naturellement :

```
(x, y, z) = u
```

ou même

```
x, y, z = u
```

Maintenant x vaut 11, y vaut 21 et z vaut 31.

Les deux affectations `x = y` et `y = x` dans

```
(x, y) = (y, x)
```

sont réalisées *en parallèle* ; il s'agit donc d'un échange (*swap*) du contenu des deux variables x et y.

7.1.2 Opérations sur les tuplets

Les opérations que l'on peut effectuer sur des tuplets sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuplets ne sont pas modifiables. Mais lorsqu'un tuplet contient comme élément une séquence modifiable (p. ex. une liste), celle-ci reste modifiable.

```
>>> a = [1, 2, 3]
>>> b = (a, 4)
>>> print(b)
 ([1, 2, 3], 4)
```

Le résultat est un tuplet à deux éléments, c'est-à-dire un couple, dont le premier élément est une liste à trois éléments. On ne peut pas modifier le premier élément de b :

```
>>> b[0] = 6
```

provoque une erreur. Par contre, on peut modifier un élément de la liste formant le premier élément de b :

```
>>> b[0][1] = 6
>>> b
 ([1, 6, 3], 4)
```

En même temps, on a bien sûr modifié le contenu de a, car a et `b[0]` pointent vers le même objet en mémoire.

```
>>> print(a)
 [1, 6, 3]
```

7.1.3 Application : vecteurs et matrices

Les vecteurs vus en mathématiques peuvent très bien être représentés par des tuplets ou par des listes de nombres. Si u et v sont deux variables de ce type, représentant des vecteurs de dimension 2, leur produit scalaire dans une base orthonormée est obtenu par

```
u[0] * v[0] + u[1] * v[1]
```

et le déterminant en dimension 2 se calcule comme suit :

```
u[0] * v[1] - u[1] * v[0]
```

Par analogie, on obtient en dimension 3 pour le produit scalaire :

```
u[0] * v[0] + u[1] * v[1] + u[2] * v[2]
```

et pour le produit vectoriel le vecteur écrit ici comme tuplet (triplet) :

```
(u[1]*v[2] - u[2]*v[1], u[2]*v[0] - u[0]*v[2], u[0]*v[1] - u[1]*v[0])
```

Pour les matrices, on construit des tuplets de tuplets, ou des listes de listes. Voici par exemple deux matrices carrées d'ordre 3 :

```
a = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
b = [[1, 3, 5], [2, 4, 6], [3, 5, 7]]
```

Pour effectuer le produit matriciel $a \cdot b$, il faut d'abord disposer d'une matrice (c'est-à-dire d'une liste de listes) de bonnes dimensions (3×3), dont tous les éléments valent 0 :

```
p = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

On pourrait également écrire, de façon moins répétitive, mais plus compliquée :

```
p = [x[:] for x in [[0] * 3] * 3]
```

Cela peut paraître bizarre, mais il faut se rendre compte que la méthode plus simple

```
p = [[0] * 3] * 3
```

ne fonctionne tout simplement pas : le vecteur-ligne `[0] * 3` est créé, puis triplé, ce qui veut dire que la pseudo-matrice p pointe trois fois vers le **même** vecteur-ligne. Lorsqu'on modifie un constituant de ce vecteur, la nouvelle valeur apparaît **trois fois** dans la colonne respective de la matrice.

Pour résoudre ce problème, on a donc construit d'abord trois vecteurs composés chacun de trois composants nuls ; le résultat est **copié** dans p, pour que les trois vecteurs-lignes (représentés par x) deviennent **indépendants** et ne représentent plus un seul objet dans la mémoire.

Ensuite on réalise le calcul $p_{ij} = \sum_k a_{ik} b_{kj}$:

```
for i in range(3):
    for j in range(3):
        for k in range(3):
            p[i][j] += a[i][k] * b[k][j]
```

Finalement, p contient le produit $a \cdot b$:

```
>>> print(p)
 [[14, 26, 38], [20, 38, 56], [26, 50, 74]]
```

7.1.4 Comment copier un tuplet ou une liste

On peut copier trivialement un tuplet t dans la variable u en écrivant :

```
u = t
```

Pour copier une liste `my_list` au premier niveau (*first-level copy*) dans une variable `copy_list`, on écrit :

```
copy_list = my_list[:]
```

Si la liste à copier est constituée éléments qui sont eux-mêmes des listes (ou d'autres structures de données modifiables), il faut effectuer une copie au deuxième niveau (*second-level copy*) en écrivant par exemple :

```
copy_list = [ x[:] for x in my_list ]
```

Ici x parcourt les éléments au premier niveau de la liste (pour une matrice il s'agit des différentes lignes qui la composent) et chacun de ces éléments (qui sont des listes dont les éléments ne sont plus des listes par hypothèse) est copié par la technique du premier niveau `x[:]`.

Une autre manière d'obtenir une copie véritable d'un objet (même pour des objets où les éléments sont imbriqués à des niveaux plus profonds que dans les matrices) consiste à utiliser la fonction de **copie profonde** :

```
from copy import deepcopy
copy_list = deepcopy(my_list)
```

Tous les éléments imbriqués dans `copy_list`, à n'importe quel niveau de profondeur, sont alors des copies véritables et indépendantes des éléments de `my_list`.

7.1.5 Exercices

Exercice 7.1 Écrire un programme qui lit un entier $n \geq 2$ et ensuite lit (ou génère de manière automatisée à l'aide de valeurs aléatoires) une matrice carrée d'ordre n. Le programme calculera et affichera alors le déterminant de cette matrice.

Pour $n = 2$ et $n = 3$, on peut implémenter les formules de calcul direct que les élèves utilisent dans les exercices du cours de mathématiques.

***** À partir de $n \geq 4$, on a intérêt à implémenter la méthode d'échelonnement de Gauß.

Exercice 7.2 * Écrire un programme qui lit un entier $n \geq 2$ et ensuite génère la matrice de Hilbert H d'ordre n :

$$H_{ij} = \frac{1}{i+j-1} \qquad (1 \leq i, j \leq n)$$

Ainsi, pour $n = 5$ on a :

$$H = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

Le programme calculera et affichera le déterminant de la matrice (voir l'exercice précédent), ainsi que l'inverse du déterminant. Pour différentes valeurs de n, comparer les déterminants avec ceux obtenus par les collègues en classe et avec les valeurs exactes (utiliser p. ex. WolframAlpha pour cette vérification).

Exercice 7.3 ** Écrire un programme qui lit un entier $n \geq 2$ et ensuite lit (ou génère de manière automatisée à l'aide de valeurs aléatoires) une matrice carrée d'ordre n. Le programme calculera et affichera l'inverse de la matrice de départ. Puis le programme calculera et affichera encore l'inverse de la matrice inverse précédente. Vérifier si le dernier résultat ne diffère pas (trop) de la matrice de départ et comparer les imprécisions éventuelles avec celles obtenues par les collègues en classe.

Indication, pour tester convenablement le programme : les matrices de Hilbert sont des matrices particulièrement « méchantes » à inverser...

Exercice 7.4 Écrire un programme qui lit le numérateur et le dénominateur d'une fraction et à l'aide d'une fonction `simplify` fournit la fraction simplifiée. Utiliser des tuplets pour représenter les fractions.

Exercice 7.5 Compléter le programme précédent : l'utilisateur peut entrer deux fractions, et après simplification de ces fractions, le programme calcule la somme, la différence, le produit et le quotient des deux fonctions fournies et affiche tous les résultats sous forme de fractions irréductibles.

7.2 Dictionnaires

7.2.1 Problème introductif

On se propose de réaliser un annuaire téléphonique qui permet notamment de retrouver rapidement le numéro de téléphone à partir du nom de l'abonné.

On pourrait réaliser deux listes en parallèle :

```
subscribers = ["LAML", "LCD", "AL"]
numbers = ["26043211", "268071", "440249−6100"]
```

Ainsi l'abonné `subscribers[i]` possède le numéro `numbers[i]`. Cette méthode présente plusieurs inconvénients :

- ⋆ il faut veiller à la synchronisation permanente des deux listes lors de modifications ;
- ⋆ lorsqu'on veut trier la liste des abonnés, il ne faut pas oublier d'opérer les mêmes changements d'ordre dans la liste des numéros (et cela ne revient **pas** à trier simplement la liste des numéros !) ;
- ⋆ l'accès à un numéro à partir du nom de l'abonné peut se faire comme suit :

```
>>> print(numbers[subscribers.index["LCD"]])
 268071
```

  mais la notation est lourde et l'exécution est assez lente, parce que la liste des abonnés devra être lue en moyenne à moitié jusqu'à ce que l'indice correct soit trouvé.

Une variable de type *dictionnaire* (en Python : type `dict`) résout ce problème de façon élégante :

```
>>> phonebook = {"LAML": "26043211", "LCD": "268071", "AL": "440249−6100"}
>>> print(phonebook["AL"])
 440249−6100
```

On peut rajouter des données par affectation :

```
phonebook["LCE"] = "7287151"
```

La liste des abonnés est obtenue par la méthode `keys`, après conversion explicite en une liste :

```
>>> print(list(phonebook.keys()))
 ['LAML', 'LCD', 'AL', 'LCE']
```

On remarque que l'ordre des éléments de cette liste n'est pas l'ordre lexicographique, et avec certains interpréteurs de Python il pourrait même différer de l'ordre d'entrée des données ! Cela est dû à l'encodage des données en mémoire, qui est plus efficace si aucun ordre précis n'est prescrit. L'ordre lexicographique est obtenu par

```
>>> print(sorted(phonebook.keys()))
 ['AL', 'LAML', 'LCD', 'LCE']
```

7.2.2 Création de dictionnaires

Les dictionnaires ressemblent aux listes et sont *mutables* (modifiables) comme elles, avec tous les effets secondaires concernant l'affectation et le passage argument ⇒ paramètre lors des appels de fonctions. Mais contrairement aux listes, les éléments enregistrés dans un dictionnaire ne sont pas disposés dans un ordre immuable dans la mémoire. En revanche, on pourra accéder à n'importe lequel d'entre eux à l'aide d'un indice spécifique appelé « clé » (*key*), qui pourra être alphabétique, numérique, ou même d'un type composé (par exemple un tuplet de nombres représentant des coordonnées).

Comme dans une liste, les éléments stockés dans un dictionnaire peuvent être de n'importe quel type, par exemple des valeurs numériques, des chaînes, des listes, des tuplets, des dictionnaires.

Puisque le type `dict` est un type modifiable, on peut créer un dictionnaire vide, puis le remplir petit à petit. Les délimiteurs de dictionnaires sont des accolades `{}`. Voici, à titre d'exemple, un dictionnaire vide :

```
birth_year = { }
```

Pour rajouter des données, il n'est pas nécessaire d'utiliser la méthode `append`, qui d'ailleurs n'existe pas pour les dictionnaires, mais on écrit simplement des affectations :

```
>>> birth_year["Mozart"] = 1756
>>> birth_year["Beethoven"] = 1770
>>> print(birth_year["Mozart"])
 1756
>>> print(birth_year)
 {'Mozart': 1756, 'Beethoven': 1770}
```

7.2.3 Opérations sur les dictionnaires

Il n'existe pas d'opération spécifique pour rajouter des éléments à un dictionnaire, mais l'affectation standard fonctionne, comme on vient de le voir :

```
birth_year["Schubert"] = 1797
```

Pour effacer des données, on utilise l'instruction **`del`** :

```
>>> del birth_year["Mozart"]
>>> print(birth_year)
 {'Beethoven': 1770, 'Schubert': 1797}
```

Le nombre d'éléments contenus dans un dictionnaire est fourni par la fonction `len` :

```
>>> print(len(birth_year))
 2
```

Pour tester si un dictionnaire contient une certaine clé, on utilise l'opérateur **`in`** :

```
if "Eilish" in birth_year:
    print(birth_year["Eilish"])
else:
    print("Musicien␣inconnu")
```

Cet extrait de programm affiche le message « `Musicien inconnu` ».

Il ne faut pas accéder à une valeur dont la clé n'existe pas dans le dictionnaire :

```
print(birth_year["Eilish"])
```

provoque une erreur de type *KeyError*. Par contre, on peut utiliser la méthode `get` pour combiner le test d'existence d'une clé et l'accès à sa valeur associée :

```
>>> print(birth_year.get("Beethoven", "Musicien␣inconnu"))
 1770
>>> print(birth_year.get("Eilish", "Musicien␣inconnu"))
 Musicien inconnu
```

On peut extraire des parties de données contenues dans un dictionnaire par les méthodes suivantes :

```
>>> print(birth_year.keys())
 dict_keys(['Beethoven', 'Schubert'])
```

Ce résultat est utilisable dans la définition de boucles :

```
>>> for m in birth_year.keys():
        print(m, "est␣né␣en", birth_year[m])
 Beethoven est né en 1770
 Schubert est né en 1797
```

Par abus de notation, le même résultat est obtenu en écrivant simplement :

```
>>> for m in birth_year:
        print(m, "est␣né␣en", birth_year[m])
 Beethoven est né en 1770
 Schubert est né en 1797
```

Si l'on préfère extraire les clés sous forme de liste ou de tuplet, on pourra écrire :

```
>>> print(list(birth_year.keys()))
 ['Beethoven', 'Schubert']
>>> print(tuple(birth_year.keys()))
 ('Beethoven', 'Schubert')
```

De manière analogue, on extrait les valeurs associées aux clés avec la méthode `values` :

```
>>> print(birth_year.values())
 dict_values([1770, 1797])
>>> print(list(birth_year.values()))
 [1770, 1797]
```

La méthode `items` renvoie tout le contenu du dictionnaire sous forme d'une séquence de couples :

```
>>> print(birth_year.items())
 dict_items([('Beethoven', 1770), ('Schubert', 1797)])
```

Ceci nous offre encore une autre manière de parcourir un dictionnaire dans une boucle :

```
>>> for m, y in birth_year.items():
        print(m, "est␣né␣en", y)
 Beethoven est né en 1770
 Schubert est né en 1797
```

Il faut toujours se rappeler que l'ordre dans lequel les éléments sont extraits du dictionnaire est imprévisible, car un dictionnaire n'est pas une séquence. Pour la même raison, on ne peut pas extraire un « sous-dictionnaire » d'un dictionnaire avec la méthode des sous-intervalles (*slicing*) applicable aux listes et aux chaînes de caractères.

Tout comme les listes, les dictionnaires ne sont pas copiés, lorsqu'on affecte une variable de dictionnaire à une autre variable (notation `d2 = d1`), mais un alias est créé qui pointe vers le même objet en mémoire que le dictionnaire de départ. Nous avions résolu le problème de la copie véritable de listes en écrivant `d2 = d1[:]`, mais cette notation (*slicing*) n'existe pas pour les dictionnaires. Pour réaliser une copie au premier niveau (*first-level copy*) d'un dictionnaire, qui sera dans la suite indépendante de l'original, il faut utiliser la méthode `copy` :

```
d2 = d1.copy()    # first-level copy
```

7.2.4 Exercice

Exercice 7.6 Écrire un programme qui peut transformer un nombre entré en notation décimale habituelle en notation romaine (avec les lettres M, D, C, L, X, V, I). Par exemple, le programme devra pouvoir transformer 2019 en `MMXIX` et 844 en `DCCCXLIV`.

Compléter ce programme pour qu'il puisse également transformer un nombre écrit en notation romaine en notation décimale habituelle. Par exemple, le programme devra pouvoir transformer `DCCCXLIV` en 844 et `MCMXCIX` en 1999. Il devra reconnaître des entrées non conformes aux standards habituels et afficher une erreur ; par exemple l'entrée `MIM` ne correspond pas à 1999, mais doit produire une erreur.

On pourra se limiter aux nombres compris entre 1 et 5000.

7.3 Exercices de synthèse

Les exercices suivants sont en général plus difficiles que ceux vus jusqu'à présent. Soit il faut appliquer plusieurs techniques de programmation vues dans les chapitres précédents, soit il faut implémenter des règles de jeu ou une stratégie gagnante non triviales.

Exercice 7.7 – La conjecture de Collatz *

Écrire un programme qui permet d'étudier cette conjecture.

Énoncé : Soit u_1 un nombre naturel non nul donné. On calcule pour $i = 1, 2, 3, \ldots$

$$u_{i+1} = \begin{cases} \frac{1}{2} \cdot u_i & \text{si } u_i \text{ est pair} \\ 3u_i + 1 & \text{si } u_i \text{ est impair} \end{cases}$$

et on s'arrête dès qu'on obtient $u_p = 1$ pour un indice p naturel.

Exemple : pour $u_1 = 14$, on obtient successivement les valeurs 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, et donc la suite se termine avec $p = 18$.

La conjecture de Collatz affirme que pour n'importe quel nombre naturel non nul u_1, la suite atteint finalement la valeur 1 et se termine ainsi.

Marche à suivre :

1. Écrire une fonction `f(n)` qui calcule et renvoie $\frac{n}{2}$ si n est pair, et $3n + 1$ si n est impair.
2. Écrire une fonction `collatz(n)` qui à partir de $u_1 = n$ donné détermine et renvoie l'indice p, c'est-à-dire l'indice du terme de la suite égal à 1. Par exemple, `collatz(14)` renvoie 18 comme réponse, parce que le 18^{e} terme de la suite vaut 1.
3. Écrire une fonction `max_collatz(n)` qui détermine la valeur maximale de p parmi toutes les suites générées pour $u_1 = 1, 2, \ldots, n$.
4. Écrire un programme principal qui demande à l'utilisateur d'entrer un nombre naturel non nul n et qui affiche les résultats suivants :
   - ⋆ la longueur maximale des suites considérées, c'est-à-dire `max_collatz(n)` ;
   - ⋆ la valeur maximale calculée par la fonction `f`, lors des calculs précédents ;
   - ⋆ un tableau de 10 lignes et de 10 colonnes bien alignées, représentant l'histogramme suivant : combien de fois la fonction `collatz` a-t-elle renvoyé comme réponse la valeur $1, 2, \ldots, 100$ lors des calculs précédents ?

Voici deux exemples d'exécution du programme :

```
Entrez n : 100
Longueur maximale : 119
Valeur maximale atteinte : 9232
   1    1    1    1    1    2    2    3    3    5
   2    3    2    2    5    2    4    6    3    5
   2    3    5    1    3    3    1    3    0    1
   3    0    2    0    1    2    0    0    0    0
   0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0
   0    0    1    0    0    0    0    0    0    0
```

```
Entrez n : 1000
Longueur maximale : 179
Valeur maximale atteinte : 250504
   1    1    1    1    1    2    2    4    4    6
   5    7    9    8   12   17   12   16    9   15
  22   11   18   22   14   23    9   17   30   10
  23    6   13   25    7   14   22    8   16    4
  10   22    7   15    3    9   21    5   11   15
   4   10    3    8   15    3    8    0    3   10
   1    5    0    1    7    1    2    4    2    6
   2    3    6    1    3    0    1    3    1    2
   1    2    3    1    2    6    2    5    2    3
   6    2    4    3    3    7    2    5   10    2
```

Exercice 7.8 – Le jeu de Tic-tac-toe *

Écrire un programme qui assure l'arbitrage du jeu de Tic-tac-toe entre deux joueurs humains.

Règles du jeu : Deux joueurs s'affrontent. Ils doivent remplir chacun à leur tour une case d'une grille 3×3, vide au départ, avec le symbole qui leur est attribué : `X` ou `O`. Le gagnant est celui qui arrive à aligner trois symboles identiques, horizontalement, verticalement ou en diagonale. Lorsque la grille est remplie et qu'aucun joueur n'a su aligner trois symboles identiques, la partie est remise.

Marche à suivre :

1. Écrire une fonction `print_board(b)` qui affiche la table de jeu `b`. Sans compter les caractères formant les bords, l'intérieur de chaque case est composé de 3 lignes et 5 colonnes de caractères. Voici un exemple :

```
    A     B     C
        |     |
1   X   |  O  |
        |     |
  -----+-----+-----
        |     |
2   X   |  X  |  O
        |     |
  -----+-----+-----
        |     |
3   X   |     |  O
        |     |
```

2. Écrire une fonction `victory(b, p)` qui analyse si le joueur `p` a trois symboles alignés dans la table de jeu `b`, et renvoie comme résultat `True` ou `False`.

3. Écrire le code principal pour l'arbitrage : les joueurs sont invités à tour de rôle par un message « Joueur 1 : » ou « Joueur 2 : » à entrer les coordonnées de leur prochain coup, sous forme de string à 2 caractères : le 1er caractère détermine la colonne (`A`, `B`, `C`, mais on acceptera aussi les lettres `a`, `b`, `c`) et le 2e caractère détermine la ligne (1, 2, 3) de la case choisie. Si cette case existe et est libre, le symbole du joueur y est placé. Voici une suite possible de coups qui aurait mené à l'affichage précédent : `B2`, `b1`, `a1`, `C3`, `a2`, `c2`, `a3`

4. Écrire le code final, exécuté lorsque la partie se termine. Le programme affichera l'un des trois messages suivants : « Joueur 1 a gagné ! », « Joueur 2 a gagné ! », « Partie remise. »

Exercice 7.9 – Le jeu de Puissance 4 (*Vier gewinnt*) *

Modifier le programme précédent pour qu'on puisse jouer non pas le jeu de Tic-tac-toe, mais le jeu de « Puissance 4 ».

Voir `https://fr.wikipedia.org/wiki/Puissance_4`

Voici le début d'un exemple d'exécution du programme. On y voit qu'il suffit d'entrer le symbole (la lettre) de la colonne où l'on désire jouer (de `A` à `G`, et par convention on accepte aussi les lettres minuscules de `a` à `g`).

```
    A B C D E F G
6 :
5 :
4 :
3 :
2 :
1 :

Joueur 1 : d
Le joueur 1 joue D1

    A B C D E F G
6 :
5 :
4 :
3 :
2 :
1 :       X

Joueur 2 : h
Joueur 2 : G
Le joueur 2 joue G1

    A B C D E F G
6 :
5 :
4 :
3 :
2 :
1 :       X     O
```

```
Joueur 1 : g
Le joueur 1 joue G2

    A B C D E F G
6 :
5 :
4 :
3 :
2 :             X
1 :       X     O
```

Exercice 7.10 – Le jeu de Nim *

Écrire un programme qui permet à l'utilisateur de jouer le « jeu de Nim » contre l'ordinateur.

Règles du jeu : Soit n un nombre entier aléatoire entre 20 et 100. Au début du jeu, il y a un tas de n allumettes et les deux joueurs peuvent retirer à tour de rôle 1, 2 ou 3 allumettes du tas. Le joueur forcé à retirer la dernière allumette perd le jeu.

Le programme assure d'un côté l'arbitrage du jeu ; ainsi il vérifie que le joueur retire à chaque coup un nombre convenable d'allumettes. D'autre part il joue lui-même de façon optimale : il gagnera la partie dès qu'il possède une stratégie gagnante, mais lorsqu'il a une position perdante, il joue 1, 2 ou 3 allumettes au hasard (pour induire le joueur humain en erreur).

Exercice 7.11 – Le jeu de Mastermind *

Lire les règles du jeu sur `https://de.wikipedia.org/wiki/Mastermind_(Spiel)`

Implémenter la table du jeu sous forme de liste contenant soit des listes à 4 éléments soit des quadruplets représentant les codes de couleurs. Une même couleur peut apparaître plusieurs fois dans le code ; en tout il y a 6 couleurs disponibles.

Après le démarrage du programme l'ordinateur déterminera un code de couleurs aléatoire qu'il n'affichera pas. L'utilisateur essayera de deviner ce code secret et le programme donnera chaque fois les indications (pions noirs et pions blancs) sur l'(in)exactitude du code.

Si l'utilisateur réussit à deviner le code secret après 10 essais au maximum, il gagne la partie, sinon il la perd.

Exercice 7.12 ** Compléter le programme précédent afin de permettre aussi à l'ordinateur de deviner le code secret choisi par l'utilisateur :

L'algorithme du programme doit être suffisamment raffiné pour pouvoir déterminer n'importe quel code secret (4 pions, 6 couleurs) en 10 essais au maximum. (Autrement dit, l'ordinateur doit toujours gagner sa partie.)

Voici un exemple d'exécution du programme, où le joueur doit d'abord deviner le code (exercice précédent) et l'ordinateur essaiera ensuite de deviner le code choisi par l'utilisateur.

```
Votre essai n° 1
Entrez votre code sous forme de 4 lettres juxtaposées : abba
Ma réponse :
Votre essai n° 2
Entrez votre code sous forme de 4 lettres juxtaposées : cdcd
Ma réponse : *.
Votre essai n° 3
Entrez votre code sous forme de 4 lettres juxtaposées : eeee
Ma réponse : *
Votre essai n° 4
```

```
Entrez votre code sous forme de 4 lettres juxtaposées : cccc
Ma réponse : ***
Votre essai n° 5
Entrez votre code sous forme de 4 lettres juxtaposées : ccec
Ma réponse : **..
Votre essai n° 6
Entrez votre code sous forme de 4 lettres juxtaposées : eccc
Ma réponse : ****
Vous avez gagné après 6 essais !
C'est mon tour maintenant, choisissez un code secret...
Mon essai n° 1 : EADA
Entrez l'évaluation sous forme 'n b' (pions noirs resp. blancs) : 0 2
Mon essai n° 2 : FDAF
Entrez l'évaluation sous forme 'n b' (pions noirs resp. blancs) : 0 3
Mon essai n° 3 : AFCD
Entrez l'évaluation sous forme 'n b' (pions noirs resp. blancs) : 2 0
Mon essai n° 4 : AFFE
Entrez l'évaluation sous forme 'n b' (pions noirs resp. blancs) : 4 0
Youpi !
```

Exercice 7.13 ** Reconsidérer le programme de l'exercice précédent. Tester l'efficacité de l'algorithme implémenté en l'appliquant successivement (de façon automatisée bien sûr) aux $6^4 = 1296$ codes secrets possibles et comptabiliser combien d'essais ont été nécessaires chaque fois pour déterminer le code secret. Afficher finalement la moyenne des nombres d'essais et aussi le nombre d'essais le plus élevé. Comparer ces valeurs avec celles obtenues par les collègues en classe.

Exercice 7.14 – Le jeu d'échecs *

Voir `https://fr.wikipedia.org/wiki/Échecs` (ceux qui ne connaissent pas le jeu devraient lire au moins les deux premiers chapitres « Règles du jeu » et « Notation des parties »).

Concevoir une manière intelligente de représenter un échiquier (c'est-à-dire une position de jeu) par une seule variable. Écrire une fonction qui affiche le contenu d'une telle variable de façon compréhensible à l'écran, par exemple en utilisant les abréviations internationales (anglaises).

Voici à titre d'exemple la position initiale du jeu :

```
r n b q k b n r
p p p p p p p p
. # . # . # . #
# . # . # . # .
. # . # . # . #
# . # . # . # .
P P P P P P P P
R N B Q K B N R
```

Exercice 7.15 ** Écrire une fonction qui accepte le 1$^{\text{er}}$ champ (décrivant l'emplacement des pièces) d'une chaîne de caractères du type FEN, renvoie la position du jeu dans une variable du type de l'exercice précédent et l'affiche à l'écran. Les champs 2 à 6 de la chaîne FEN complète ne sont pas fournis comme arguments.

Voir `https://fr.wikipedia.org/wiki/Notation_Forsyth-Edwards`

Exercice 7.16 – Jeu de bataille navale **

Règles du jeu : Deux joueurs placent chacun des navires sur une grille tenue secrète et tentent ensuite de toucher les navires adverses. Le gagnant est celui qui parvient à couler tous les navires de l'adversaire avant que tous les siens ne le soient. On dit qu'un navire est coulé si chacune de ses cases a été touchées par un coup de l'adversaire.

Chaque joueur possède les mêmes types de navires : un navire à 5 cases, un navire à 4 cases, deux navires à 3 cases et un navire à 2 cases.

Un joueur peut placer chaque navire soit horizontalement, soit verticalement dans sa grille, de dimension 10×10. Les coordonnées de la grille sont numérotées de 1 à 10 verticalement (du haut vers le bas) et de A à K horizontalement (en omettant la lettre I). Deux navires d'un même joueur peuvent être *tangents* (occuper des cases voisines), mais ils ne peuvent pas se croiser/superposer (occuper des cases identiques).

Voir `https://fr.wikipedia.org/wiki/Bataille_navale_(jeu)`

Marche à suivre :

1. Le programme doit être capable de placer ses cinq navires dans sa grille, conformément aux règles susmentionnées.

2. Le programme demande à l'utilisateur de placer ses navires dans un ordre donné (par exemple, suivant les longueurs 5-4-3-3-2). Il vérifie que l'utilisateur place ses navires conformément aux règles.

3. L'utilisateur et l'ordinateur jouent à tour de rôle. Le premier joueur à faire couler tous les navires de son adversaire gagne la partie.

   Pour le jeu de l'ordinateur, un algorithme trivial consiste à choisir une case au hasard qui n'a pas encore été une cible antérieure. Évidemment il existe des algorithmes beaucoup plus efficaces et subtils à programmer... avis aux amateurs !

8 Fichiers [chapitre optionnel]

8.1 Introduction

Un fichier informatique est, au sens commun, une collection de données numériques réunies sous un même nom, enregistrées sur un support de stockage permanent, appelé mémoire de masse, tel qu'un disque dur, un CD-ROM, une mémoire flash, et manipulées comme une unité. Un fichier porte un nom de fichier qui sert à désigner le contenu et à y accéder. Ce nom comporte souvent un suffixe : l'extension, qui renseigne sur la nature des informations contenues dans le fichier et donc des logiciels utilisables pour le manipuler. Chaque fichier comporte un certain nombre de métadonnées, informations telles que la longueur du fichier, son auteur, les personnes autorisées à le manipuler, ou la date de la dernière modification. Le contenu est l'essence du fichier. Il existe des centaines, voire des milliers de types de fichiers, qui se différencient par la nature du contenu, le format, le logiciel utilisé pour manipuler le contenu, et l'usage qu'en fait l'ordinateur. La nature du contenu peut être des textes, des images, de l'audio ou de la vidéo.

Le format de fichier est la convention selon laquelle les informations sont numérisées et organisées dans le fichier et sert d'emballage dans lequel sera mis le contenu ainsi que les métadonnées. L'extension lorsqu'elle est présente, suffixe un nom du fichier, afin de renseigner sur le format du fichier et donc sur les logiciels pouvant être utilisés pour le manipuler. Chaque fichier peut être enregistré n'importe où dans le système de fichiers, et le logiciel qui le manipule propose un emplacement conventionnel de stockage. Certains formats sont dits propriétaires, c'est-à-dire que le format n'est connu que de son auteur et n'a jamais fait l'objet de publications.

Dans le traitement informatique des fichiers, on distingue deux grandes catégories :

- ⋆ les fichiers-textes dont le contenu est lisible lorsqu'on l'affiche dans un éditeur de texte ou dans un navigateur internet et
- ⋆ les fichiers binaires. Dans ces derniers, les informations y sont stockées de manière particulière et il faut utiliser un programme spécial pour accéder à leur contenu (image JPG, document Word, etc.)

Nom	**Nature du contenu**
exécutables	fichiers qui peuvent être exécutés par l'ordinateur – autrement dit des programmes.
compressés	fichiers codés selon un procédé qui les rend plus petits que les fichiers originaux non codés. Un programme décompresseur est nécessaire pour effectuer le décodage inverse et retrouver ainsi le fichier original.
images/audio/vidéo	des fichiers qui contiennent des images et du son sous une forme exploitable par l'ordinateur. De tels fichiers peuvent contenir des photos, des pictogrammes, des graphiques, des chansons, de la musique, des émissions radio ou des films.
documents	documents écrits, destinés à être imprimés et lus. Le fichier contient le texte ainsi que les informations de typographie (polices de caractères, couleurs).
textes	les fichiers de texte brut contiennent des textes écrits, sans indications de typographie. Il peut s'agir de textes destinés aux utilisateurs, tels que des modes d'emploi ou des brouillons ; ou alors de textes destinés à l'ordinateur tels que du code-source ou bien des données pour un programme.

Dans ce qui suit, nous ne considérons que les fichiers-textes.

8.2 Manipulation de fichiers-textes

En Python, il est évidemment possible de travailler avec des fichiers-textes : il faut définir une variable associée à ce fichier en utilisant la fonction `open(file_name, mode)`.

`file_name` désigne une chaîne de caractères contenant le nom du fichier tel qu'il est connu par le système d'exploitation (par exemple sur le disque dur). Le paramètre `mode` indique la façon dont nous voulons travailler avec ce fichier. Voici quelques valeurs possibles pour le mode et les significations respectives.

- ⋆ mode `"r"` (*read*) : ouvre le fichier en mode lecture. C'est le mode par défaut.
- ⋆ mode `"w"` (*write*) : ouvre le fichier en mode écriture. Si le fichier existe déjà, il est vidé (remplacé). Dans le cas contraire, il est créé.
- ⋆ mode `"a"` (*append*) : ouvre le fichier en mode écriture/ajout. Les données seront ajoutées à la fin du fichier. Si le fichier n'existe pas encore, il est créé.

Il existe plusieurs méthodes pour lire et écrire dans un fichier. Dans ce cours, nous n'en considérons que deux : les méthodes `readline` et `write` peuvent être appliquées à une variable de type fichier.

La méthode `readline` renvoie une chaîne de caractères contenant l'entièreté de la ligne lue, le caractère fin de ligne compris. Lorsque la fin du fichier est atteinte, cette méthode retourne la chaîne de caractères vide `""`.

La méthode `write` écrit dans le fichier l'argument du type string fourni.

Lorsque le traitement du fichier est terminé, il faut fermer le fichier à l'aide de la méthode `close`.

Voici un exemple :

```
file = open("ages.txt", "w")
name = input("Enter␣the␣name:␣")
while (name != ""):
    age = input("Enter␣the␣age:␣")
    file.write(name + "␣" + age + "\n")
    name = input("Enter␣the␣name:␣")
file.close()

file = open("data.txt", "r")
line = file.readline()
while line != "":
    print(line, end = "")
    line = file.readline()
file.close()

file = open("data.txt", "a")
name = input("Enter␣the␣name:␣")
while (name != ""):
    age = input("Enter␣the␣age:␣")
    file.write(name + "␣" + age + '\n')
    name = input("Enter␣the␣name:␣")
file.close()

file = open("data.txt", "r")
line = file.readline()
```

```
while line != "":
    print(line, end = "")
    line = file.readline()
file.close()
```

Voici un exemple d'exécution du programme :

```
Enter the name: Tintin
Enter the age: 77
Enter the name: Astérix
Enter the age: 30
Enter the name:
Tintin 77
Astérix 30
Enter the name: Obélix
Enter the age: 40
Enter the name:
Tintin 77
Astérix 30
Obélix 40
```

8.3 Exercices

Exercice 8.1 Écrire un programme qui détermine le nombre de lignes et de mots (séparés par au moins une espace) dans un fichier-texte dont le nom est entré par l'utilisateur.

Exercice 8.2 Un commissaire de gouvernement en charge d'un examen officiel dispose du fichier de toutes les notes des élèves dans une matière donnée. La correction de l'examen est toujours effectuée par exactement deux correcteurs dont les notes sont reprises en désordre dans le fichier `notes.txt`. Pour garantir l'impartialité des corrections, les correcteurs ne connaissent pas les noms des élèves, mais uniquement un numéro qui leur a été attribué. Voici un exemple de fichier `notes.txt` :

```
2 40
4 35
2 38
1 25
3 45
6 45
3 50
1 28
6 58
4 38
```

Le but du programme est d'afficher la moyenne des élèves en ordre croissant dans cette matière. Pour le fichier de données ci-dessus, le résultat à produire aura l'aspect suivant :

```
Numéro élève -- Moyenne
  1              26.5
  2              39.0
  3              47.5
  4              36.5
  6              51.5
```

Exercice 8.3 Soit un fichier d'entiers représentant les cours en bourse d'un titre sur une période. Écrire un programme qui affiche le nombre de fois que le titre a effectué une hausse de cours d'un jour à l'autre. Pour simplifier, nous supposons que les cours de bourse sont représentés par des entiers. Les valeurs dans le fichier peuvent être réparties sur plusieurs lignes.

Exemple de fichier (se limitant à 7 jours consécutifs) :

```
45 47 51 38 45
57 57
```

Le programme affiche : `4`

Le titre a en effet subi 4 hausses d'un jour à l'autre (45 $\nearrow$ 47, 47 $\nearrow$ 51, 38 $\nearrow$ 45, 45 $\nearrow$ 57).

Exercice 8.4 Écrire un programme qui permet d'afficher les résultats d'une élection. Lors de cette élection, au maximum 10 listes et 100 candidats se présentent devant les électeurs.

On dispose d'un fichier `election.txt` contenant toutes les données de cette élection. Il est structuré de la manière suivante :

- ⋆ Le premier entier représente le nombre de listes.
- ⋆ Il est suivi d'une série d'entiers qui fournit le nombre de candidats. Le n-ième entier de cette liste représente le numéro de la liste sur laquelle ce n-ième candidat se présente.
- ⋆ Cette liste de candidats est suivie de tous les bulletins de vote, c'est-à-dire des entiers représentant les numéros des candidats ayant reçu une voix.

Le programme affichera une statistique sur le nombre et le pourcentage de voix obtenus par chacune des listes en présence.

Voici un exemple de fichier `election.txt` :

```
3
1 2 2 3 3 3 2
1 1 2 2 2 5 5 1 6 1
4 1 2 3 3 3 1
```

Dans cet exemple, il y a 7 candidats avec pour chacun le numéro de la liste sur laquelle il est inscrit. Ici, le candidat 1 est sur la liste 1, le candidat 2 sur la liste 2, le candidat 3 sur la liste 2, le candidat 4 sur la liste 3, etc.

Les deux dernières lignes représentent les bulletins de vote. Leur nombre est inconnu à l'avance et ils sont répartis sur une ou plusieurs lignes.

Le programme à écrire produira les résultats suivants (exemple d'exécution basé sur le fichier `election.txt` ci-dessus) :

```
Candidat - Liste
   1        1
   2        2
   3        2
   4        3
   5        3
   6        3
   7        2
Liste :      1      2      3
Voix :       6      7      4
Liste  1 :  35.29%
Liste  2 :  41.18%
Liste  3 :  23.53%
```