

Pascal ORTIZ

Programmation Python

(Version du 22 novembre 2016)

Sommaire

I	Interface, variables, opérations	3
1	Le processus de programmation en Python	3
2	Utilisation	5
3	Affichage	8
4	Variables, expressions	9
5	Constantes, nombres	13
6	Les types	15
7	Opérations	17
8	Division entière	21
9	Plus d’affichage	25
10	Saisie du code	28
11	Compléments	29
	EXERCICES TYPE	33
	EXERCICES	35
II	Conditions	38
1	Booléens	38
	EXERCICES TYPE	43
	EXERCICES	49
2	Instructions conditionnelles	51
	EXERCICES TYPE	63
	EXERCICES	65
III	Boucles for, listes, boucles while	69
1	Boucles for	69
	EXERCICES	73
2	Listes	73
	EXERCICE TYPE	78
	EXERCICES	79
3	Parcourir, répéter	81
	EXERCICES TYPE	88
	EXERCICES	95
4	Les chaînes	99
	EXERCICES	106

5	Boucles imbriquées	108
	EXERCICES TYPE	110
	EXERCICES	111
6	Boucle while	115
	EXERCICES TYPE	122
	EXERCICES	124
7	Compléments	127
IV	Fonctions	132
1	Le concept de fonction	132
2	Renvoi d'une fonction	136
3	Créer, utiliser des fonctions	141
4	Divers	148
5	Exercices	158
	EXERCICES TYPE	158
	EXERCICES	164
6	Récurtivité	173
	EXERCICES TYPE	184
	EXERCICES	188
V	Calcul matriciel, exceptions	194
	EXERCICES TYPE	219
	EXERCICES	223

Table des matières

I	Interface, variables, opérations	3
1	Le processus de programmation en Python	3
a)	Le processus de programmation en Python	3
2	Utilisation	5
a)	Pouvoir utiliser un interpréteur	5
b)	Éditeur intégré	6
3	Affichage	8
a)	Afficher	8
b)	Afficher un message	9
c)	<code>print</code> et saut de ligne par défaut	9
4	Variables, expressions	9
a)	Variables	9
b)	Notion d'expression	10
c)	Syntaxe basique des noms de variables	10
d)	Nom ou variable non reconnus	11
e)	Réaffectation de variable	11
f)	Affichage multiple	12
5	Constantes, nombres	13
a)	Notions sur les constantes	13
b)	Constantes entières	13
c)	Grands entiers	14
d)	Constantes flottantes	14
6	Les types	15
a)	Types de base	15
b)	Variables et typage	16
c)	Les fonctions built-in	16
7	Opérations	17
a)	Opérations sur les nombres	17
b)	Utiliser des variables	18
c)	Division flottante	19
d)	Opérations mixtes entre flottants et entiers	19
e)	Opérateurs « égale à » et « différent de »	19
f)	Comparaison de nombres	20

	g)	Comparaison de flottants	20
	h)	Confusion entre égalité et affectation	21
8		Division entière	21
	a)	Multiples, division entière	22
	b)	Quotient et reste de division entière	23
	c)	Division entière et quotient exact	24
9		Plus d’affichage	25
	a)	Affichage amélioré	25
	b)	Afficher un message complexe	26
	c)	Affichage et forcer un passage à la ligne	27
10		Saisie du code	28
	a)	Indentation accidentelle	28
	b)	Placer des commentaires	28
11		Compléments	29
	a)	L’affectation	29
	b)	Affectation combinée	30
	c)	Partie entière et type <code>int</code>	31
	d)	La bibliothèque standard	32
	e)	Importer une fonctionnalité standard	32
		EXERCICES TYPE	33
		EXERCICES	35

II Conditions 38

1		Booléens	38
	a)	Opérateurs logiques ET, OU et NON	38
	b)	Les booléens en Python	40
	c)	Opérateurs <code>and</code> , <code>or</code> et <code>not</code>	40
	d)	Expressions booléennes	41
		EXERCICES TYPE	43
		EXERCICES	49
2		Instructions conditionnelles	51
	a)	Instruction <code>if</code>	51
	b)	Indentation d’une instruction composée	54
	c)	Instructions <code>if</code> imbriquées	55
	d)	Instruction <code>if/else</code>	56
	e)	Indentation d’une instruction composée avec clause	57
	f)	Indentation intelligente des éditeurs	58
	g)	Instruction <code>if/elif</code>	59
	h)	Suite de <code>if</code> vs suite de <code>if/elif</code>	60
	i)	Les conditions <code>if/elif</code>	61
	j)	Instruction <code>if/elif/else</code>	62
	k)	Parenthèse autour du booléen suivant un <code>if</code>	63

EXERCICES TYPE	63
EXERCICES	65

III Boucles for, listes, boucles while 69

1	Boucles for	69
a)	Boucle for : introduction	69
b)	Répéter une action n fois avec une boucle for	71
c)	Affichage sur une même ligne et boucle for	72
d)	Itérer sur des entiers consécutifs	72

EXERCICES	73
---------------------	----

2	Listes	73
a)	Notion de liste	73
b)	Indexation des éléments d'une liste	74
c)	Opérations sur les éléments d'une liste	75
d)	Nombre d'éléments d'une liste	75
e)	Appartenance à une liste	76
f)	Non-appartenance à une liste	76
g)	Modifier les éléments d'une liste	76
h)	append : adjoindre un élément à une liste	77
i)	Nature des éléments d'une liste	78

EXERCICE TYPE	78
-------------------------	----

EXERCICES	79
---------------------	----

3	Parcourir, répéter	81
a)	Boucle for : parcours d'une liste sans utiliser d'indices	81
b)	Liste construite depuis la liste vide	82
c)	Boucle for et calcul de somme	82
d)	Boucle for, filtrage et comptage	83
e)	Parcours complet d'une liste par indices	84
f)	Parcours de liste : indices voisins de l'indice courant	85
g)	Boucle for et calcul de maximum	85
h)	La technique du drapeau dans une boucle for	86

EXERCICES TYPE	88
--------------------------	----

EXERCICES	95
---------------------	----

4	Les chaînes	99
a)	Les chaînes : présentation	99
b)	La notion de chaîne littérale	100
c)	Opérations sur des chaînes	101
d)	Chaîne vide	102
e)	Concaténation de chaînes	102
f)	Répétition de chaînes	103
g)	Accès aux caractères d'une chaîne	104
h)	Dépassement d'indice dans une chaîne	105

i)	Créer une chaîne à partir de la chaîne vide	105
j)	Boucle for : parcours de chaînes	106
k)	Méthode count	106
l)	Recherche de sous-chaîne avec l'opérateur in	106
EXERCICES		106
5	Boucles imbriquées	108
a)	Boucles for imbriquées	108
b)	Effectuer plusieurs tests à l'aide d'une boucle for	109
EXERCICES TYPE		110
EXERCICES		111
6	Boucle while	115
a)	Boucle while	115
b)	Boucle for vs boucle while	118
c)	Boucle while et parcours de liste	119
d)	Boucle for vs boucle while : tableau récapitulatif	120
e)	Boucle for : break	120
f)	Boucles et affectations combinées	121
EXERCICES TYPE		122
EXERCICES		124
7	Compléments	127
a)	Concaténation de listes	127
b)	Boucle for : parcours de chaînes	128
c)	Boucle infinie	128
d)	Utilisation d'une boucle infinie	131

IV Fonctions 132

1	Le concept de fonction	132
a)	Les fonctions en Python	132
b)	Une fonction typique	133
2	Renvoi d'une fonction	136
a)	return et fin d'exécution	136
b)	Fonction renvoyant plusieurs valeurs	137
c)	Instructions return multiples	137
d)	Absence de return	138
e)	Retour d'une fonction sans instruction return	139
f)	Procédure ou pas ?	139
g)	Notion d'effet de bord	140
3	Créer, utiliser des fonctions	141
a)	Des instructions à la construction d'une fonction	141
b)	Enchaîner des fonctions	143
c)	De l'usage des fonctions	144
4	Divers	148

a)	Fonction sans paramètre	149
b)	Variables locales	150
c)	Variables globales	151
d)	Constantes placées en variables globales	152
e)	Fonction non appelée	152
f)	Le passage des arguments par affectation	153
g)	Modification par une fonction d'un de ses arguments	154
h)	Boucle for et return	156
i)	Le Hasard en Python	156
j)	Fonction randrange	157
k)	Fonction randint	158
5	Exercices	158
	EXERCICES TYPE	158
	EXERCICES	164
6	Récurtivité	173
a)	Introduction à la récurtivité	173
b)	Outil de visualisation des appels récurtifs	174
c)	Ecrire un algorithme récurtif	174
d)	Résolution de problème par des algorithmes récurtifs	175
e)	Récurtivité inefficace	176
f)	Récurtivité illustrée par la conversion en base b	177
g)	Récurtivité illustrée par la recherche dichotomique	179
h)	Récurtivité illustrée par le produit cartésien	181
i)	Limitation du nombre d'appels récurtifs	184
	EXERCICES TYPE	184
	EXERCICES	188

V Calcul matriciel, exceptions 194

a)	Notion de matrice	194
b)	Comment implémenter des matrices en Python ?	194
c)	Nombre de lignes et nombre de colonnes	195
d)	Parcours d'une matrice	196
e)	Quelques matrices remarquables	196
f)	Somme, opposé, produit externe, transposée, trace	198
g)	Bibliothèque de fonctions matricielles	199
h)	Fonctionnalités importées depuis un fichier	199
i)	La fonction d'affichage d'une matrice	200
j)	Fonctions de la bibliothèque : exemples	201
k)	Tester son code avec des matrices	204
l)	Produit de deux matrices	205
m)	Notion d'exception	206
n)	La gestion des exceptions	208

o)	La gestion d'une exception de type donné	210
p)	L'exception <code>TypeError</code>	210
q)	Exception non gérée	211
r)	Plusieurs clauses <code>except</code>	211
s)	Le rôle d'une clause <code>except</code>	213
t)	Accéder par un nom à une exception active	214
u)	<code>except</code> <code>sec</code>	214
v)	L'exception <code>TypeError</code>	215
w)	Exception de type <code>ValueError</code>	216
x)	Exception de type <code>IndexError</code>	216
y)	Déclencher volontairement une exception	217
EXERCICES TYPE		219
EXERCICES		223

Chapitre I

Interface, variables, opérations

1 Le processus de programmation en Python

a) Le processus de programmation en Python

Le cycle de programmation

Soit à produire un programme écrit en langage Python et qui résout un problème, par exemple :

PROBLÈME : *calculer et afficher la date du prochain vendredi 13.*

On considère un programmeur qui a les compétences suffisantes pour écrire un tel programme. Le programmeur ouvre un environnement de programmation¹. Voici les étapes par lesquelles il va passer :

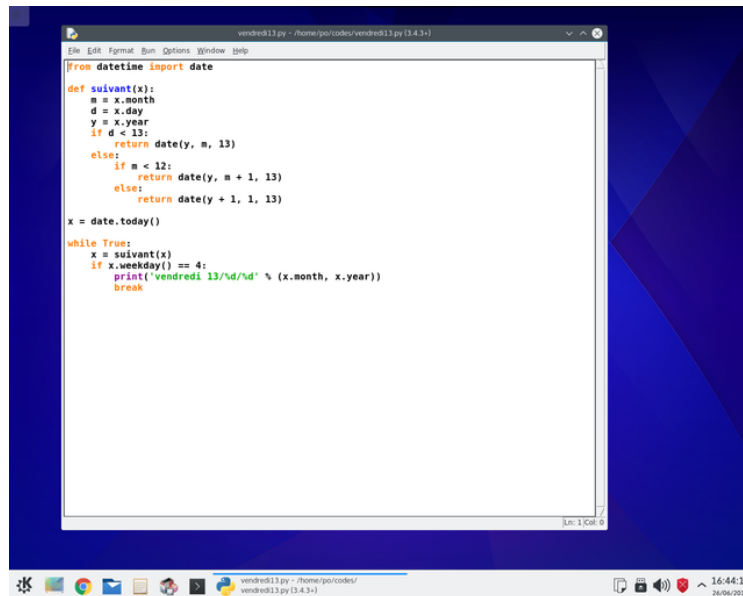
— Étape 1 : *édition du code-source*

Le programmeur ouvre un programme plus ou moins sophistiqué permettant de saisir (au clavier) le code-source dans le langage de programmation Python et censé résoudre le problème posé et il écrit ce code-source, cf. la première image ci-dessous.

— Étape 2 : *exécution du code-source*

Le programmeur exécute le code qu'il a écrit à l'étape 1 pour voir si ce code réalise bien ce pour quoi il a été écrit, ici trouver le prochain vendredi 13, cf. la deuxième image ci-dessous. Si ce n'est pas le cas, il revient à l'étape 1 et modifie le code-source.

1. ils sont de types très variés, impossible de les décrire pour l'instant.

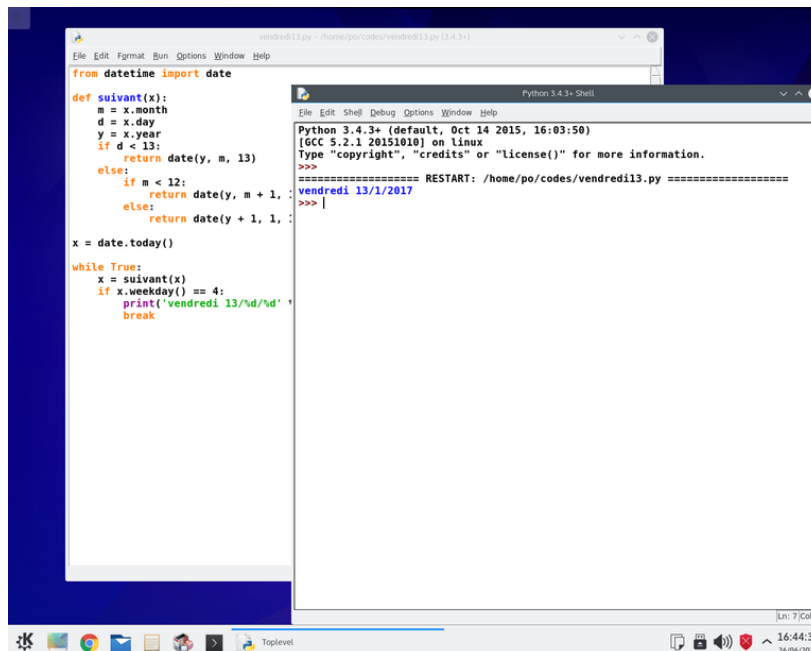


```
from datetime import date

def suivant(x):
    m = x.month
    d = x.day
    y = x.year
    if d < 13:
        return date(y, m, 13)
    else:
        if m < 12:
            return date(y, m + 1, 13)
        else:
            return date(y + 1, 1, 13)

x = date.today()
while True:
    x = suivant(x)
    if x.weekday() == 4:
        print('vendredi 13/%d/%d' % (x.month, x.year))
        break
```

Étape 1 : code Python saisi dans l'éditeur intégré IDLE



```
Python 3.4.3+ Shell

Python 3.4.3+ (default, Oct 14 2015, 16:03:50)
[GCC 5.2.1 20151010] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /home/po/codes/vendredi13.py =====
>>> vendredi 13/1/2017
>>>
```

Étape 2 : exécution du code précédent. On lit la réponse dans la fenêtre de droite :
vendredi 13/1/2017

Même pour un nouveau-venu à la programmation, le schéma en deux étapes ci-dessus reste valide.

L'étape 1 est en général la plus longue et la durée et la qualité de la réalisation de cette étape vont dépendre des connaissances plus ou moins approfondies du programmeur.

Programme du vendredi 13

Concernant le problème du prochain vendredi 13, le programmeur pourrait écrire dans un code-source tel que celui-ci :

```

vendredi13.py
1 from datetime import date
2
3 def suivant(x):
4     m = x.month
5     d = x.day
6     y = x.year
7     if d < 13:
8         return date(y, m, 13)
9     else:
10        if m < 12:
11            return date(y, m + 1, 13)
12        else:
13            return date(y + 1, 1, 13)
14
15 x = date.today()
16
17 while True:
18     x = suivant(x)
19     if x.weekday() == 4:
20         print('vendredi 13/%d/%d' % (x.month, x.year))
21         break

```

Si le code `vendredi13.py` est exécuté, en 2016, après la rentrée, le programme affichera

```

1 vendredi 13/1/2017

```

Le cycle édition-exécution

Selon que le résultat de l'étape 2 est satisfaisant ou pas, le programmeur retourne à l'étape 1 pour modifier le code-source et ainsi de suite : c'est le cycle dit *édition-exécution*.

Suivant les programmes, ce cycle peut s'étaler entre quelques secondes (pour les petits programmes) et plusieurs mois pour de programmes complexes et volumineux, faits en équipe.

2 Utilisation

a) Pouvoir utiliser un interpréteur

Pour coder en Python, il **faut** disposer d'un logiciel appelé un *interpréteur Python* qui puisse exécuter du code écrit en Python. Pour disposer d'un interpréteur, il y a deux situations :

- l'interpréteur est local, installé sur votre ordinateur
- l'interpréteur est distant, accessible via Internet depuis un site web

Dans le cadre d'un auto-apprentissage de Python, la solution de l'interpréteur local suppose, la plupart du temps, c'est **vous** qui vous chargez de l'installation de Python et aussi ... des difficultés qu'elle peut occasionner.

L'interpréteur distant a tous les avantages (et les inconvénients) d'une utilisation via une connexion Internet. Voici quelques avantages :

- vous n'avez rien à installer sur votre machine,
- l'interpréteur fonctionnera quel que soit votre système d'exploitation,

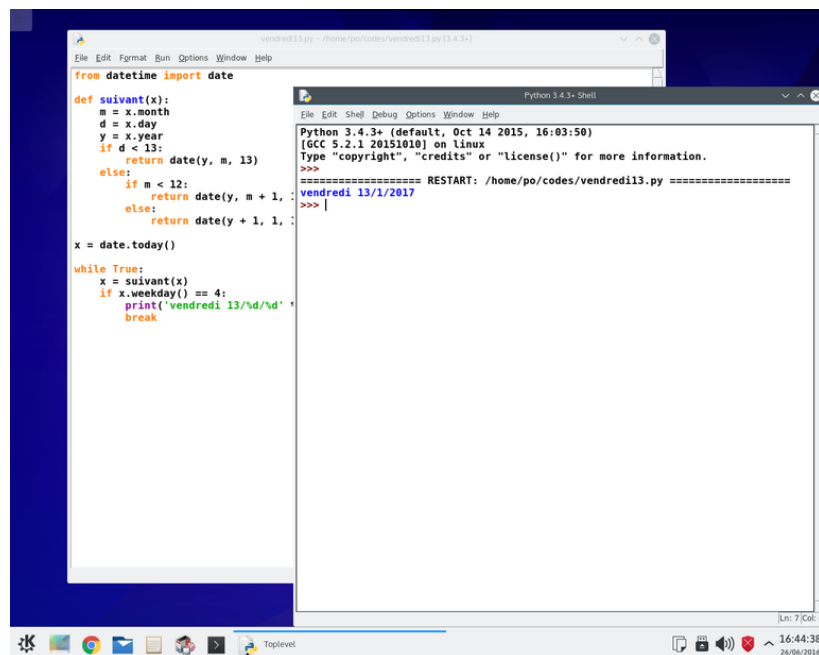
- vous pourrez coder depuis n'importe quel ordinateur connecté à Internet,
- votre interpréteur sera à jour en permanence (avec des nuances),
- votre code est sauvegardé à distance (avec des nuances),
- vous pouvez facilement partager votre code avec d'autres personnes (avec des nuances).

Pour apprendre ou enseigner Python, le choix de l'interpréteur distant est à considérer.

b) Éditeur intégré

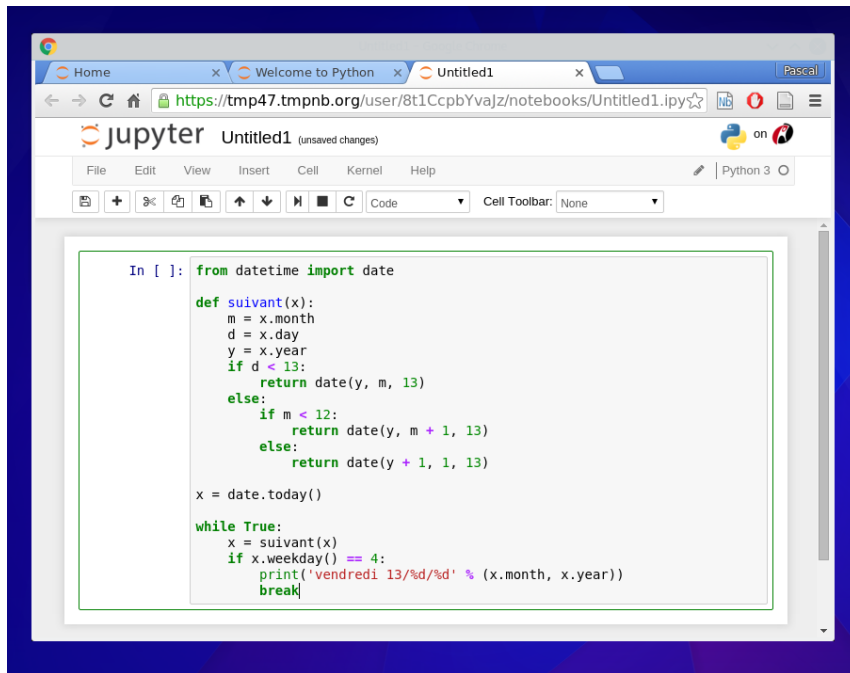
Tout environnement permettant à la fois d'éditer du code Python et de l'exécuter sera appelé *éditeur intégré*². Un éditeur intégré peut être distant ou être installé localement sur votre machine. Par exemple,

- l'éditeur IDLE (cf. [la première image ci-dessous](#)) ou l'éditeur PyCharm sont des éditeurs locaux ;
- des sites web comme [jupyter](#) (cf. [la deuxième image ci-dessous](#)) ou [tutorialspoint](#) disposent d'un éditeur intégré distant.



Un éditeur local : l'éditeur intégré par défaut IDLE

2. Le terme usuel est plutôt d'IDE pour *Integrated Development Environment* c'est-à-dire *environnement de développement intégré*.



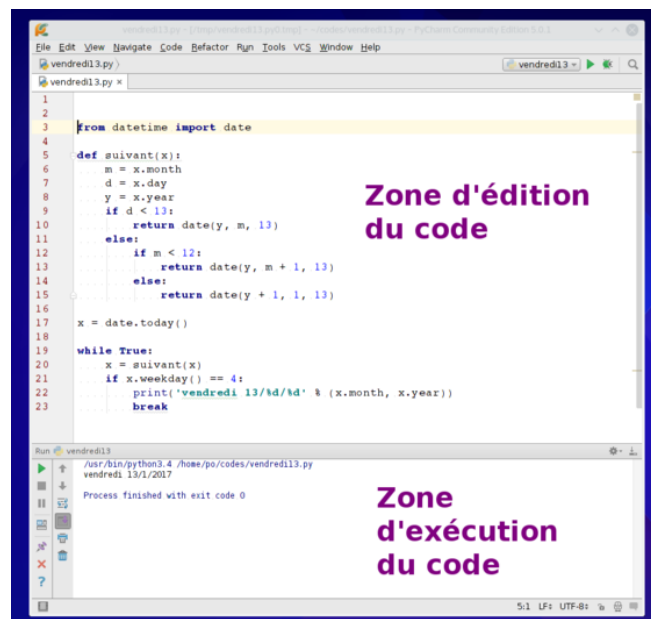
Un éditeur distant : l'éditeur intégré Jupyter disponible sur le site try.jupyter.org

Tout éditeur intégré contient 2 zones bien distinctes (cf. les images ci-après) :

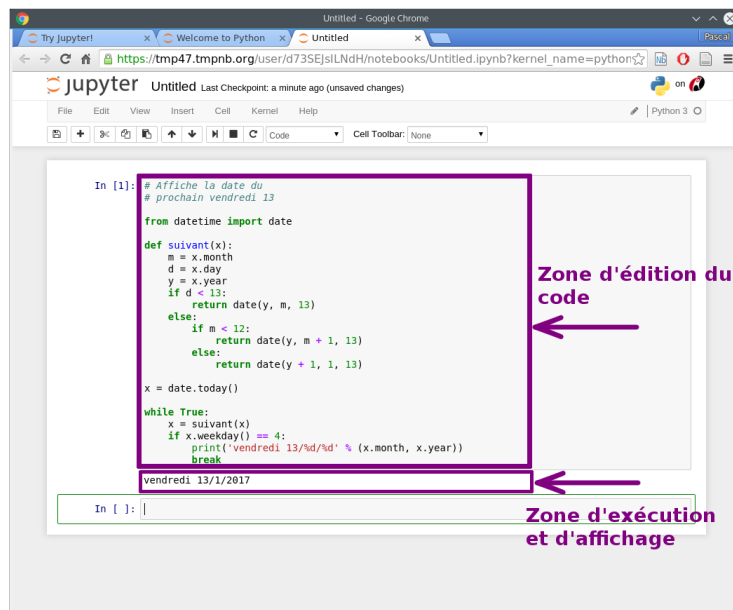
- une zone d'édition de code
- une zone de sortie

La zone d'édition est la zone de l'éditeur où l'utilisateur écrit son code Python. La zone de sortie est la zone de l'IDE où apparaissent les affichages générés par l'exécution du code de la zone éditeur.

Voici des images de quelques éditeurs intégrés montrant les deux zones :



Zones d'édition de code et d'exécution de code dans l'éditeur intégré PyCharm



Cellules d'édition de code et d'exécution de code dans l'éditeur distant Jupyter

En pratique,

- le programmeur tape son code dans la zone d'édition
- il appuie sur une touche du clavier ou clique sur un bouton ou dans un menu pour exécuter son code
- il lit le résultat de l'exécution dans la zone de sortie.

3 Affichage

a) Afficher

Soit le code dont le contenu est exactement le suivant :

```
1 5 + 5
2 2 + 2
```

Lorsqu'on demande à Python d'exécuter ce code, Python calculera $5 + 5 = 10$ puis calculera $2 + 2 = 4$. Ensuite, selon l'environnement de programmation³, le programme n'affichera rien ou n'affichera que le résultat du dernier calcul (ici 4).

Pour avoir la **garantie** que le résultat d'une opération sera affiché et correctement affiché, **il faut le demander explicitement** à Python avec une fonction fournie par le langage, la fonction **print** :

```
1 print(5 + 5)
2 print(2 + 2)
```

qui affiche

3. L'affichage décrit ici s'applique lorsque le code Python est écrit dans un fichier Python d'extension `.py` ou assimilé. L'affichage en (mode IPython)[§Afficher sous IPython§] et l'affichage en (mode interactif)[§En mode interactif, quel code produit un affichage?§] ont des particularités.

```
1 10
2 4
```

`print` est une fonction *standard* du langage Python pour signifier qu'elle est fournie nativement par le langage Python et immédiatement disponible pour utilisation. Comme c'est une fonction, pour l'utiliser et ainsi obtenir un affichage, il faut placer une paire de parenthèses après le nom `print` de la fonction et placer entre les parenthèses l'objet à afficher.

b) Afficher un message

On peut afficher un « message » avec la fonction `print`. Dans le code-source, le message est placé entre guillemets :

```
1 print("--->> Bonjour ! <<---")
2 --->> Bonjour ! <<---
```

Le message peut être plus ou moins élaboré voire fantaisiste.

c) `print` et saut de ligne par défaut

Soit le code suivant :

```
1 print(42)
2 print(421)
```

A l'exécution, la sortie affichée est

```
1 42
2 421
```

et non

```
1 42421
```

Autrement dit, l'instruction `print(message)` affiche le contenu de `message` immédiatement suivi d'un saut de ligne.

4 Variables, expressions

a) Variables

Voici un code utilisant une variable, appelée `toto` :

```
1 toto = 42
2 print(toto + 10)
3 52
```

— Ligne 1 : `toto` désigne une variable qui référence 42. La variable `toto` est comme une *étiquette* qui permet d'accéder à l'entier 42 écrit quelque part en mémoire. L'entier 42 est ce qu'on appelle un *objet* au sens d'une entité présente dans la mémoire de l'ordinateur au moment où Python examine l'instruction `toto = 42`.

— Ligne 2 :

— d'abord Python regarde s'il dispose d'une variable `toto` référencée en mémoire (c'est le

cas, cf. ligne 1) puis il récupère le contenu de cet emplacement-mémoire et effectue le calcul demandé.

- `toto + 10` est une *expression* et qui a une valeur (ici 52).
- `print` permet d'afficher la valeur de l'expression.

Un nom de variable est sensible à la casse (lettre en majuscule ou minuscule) :

```
1 year = 2000
2 Year = 3000
3 print(year)
4 print(Year)
5 2000
6 3000
```

Vocabulaire

- L'instruction `toto = 42` est une *affectation* : on *affecte* 42 à la variable `toto`.
- Quand une variable est définie pour la première fois, la première valeur qu'elle reçoit lorsqu'elle est définie s'appelle l'*initialisation*. Donc, ci-dessus, `toto` est initialisée à 42.

En Python, une variable est définie essentiellement par affectation.

La notion de variable est fondamentale en programmation et est le premier maillon vers l'abstraction. Il est capital d'écrire du code où les données sont placées dans des variables.

b) Notion d'expression

Soit le code suivant :

```
1 d = 4
2 u = 2
3 x = 10 * d + u
4
5 print(10 * x + 1)
6 421
```

On y définit des variables, par exemple `d` ou `u`, et en combinant ces variables avec des constantes (comme 10) et des opérateurs comme `*` et `+`, on obtient des « écritures » qui s'appellent des *expressions*, comme `10 * d + u` (ligne 3).

En combinant des expressions à nouveau avec des opérateurs et des constantes, on obtient de nouvelles expressions, comme ci-dessus `10 * x + 1` (ligne 5).

Par définition, une expression a toujours une *valeur*. Par exemple la valeur de l'expression `10 * x + 1` est l'entier 421. En revanche, l'écriture `d = 4` (ligne 1) n'a pas de valeur et n'est pas une expression⁴.

Pour afficher une expression `E`, on utilise l'instruction `print(E)` et qui affiche dans une sortie⁵ sous forme textuelle la *valeur* de cette expression.

c) Syntaxe basique des noms de variables

Voici un programme utilisant des noms valides de variables :

4. À la différence de langages comme C ou Java.
5. La sortie dépend de l'environnement d'utilisation (mode fichier, Jupyter, console Python, etc)

```

1 tokyo = 2020
2 tokyo2020 = 32
3 TOKYO = 2020
4 tokyoTokio = 32

```

La plupart du temps, un nom de variable est un mot formé de caractères alphabétiques, (majuscules ou minuscules) ou de chiffres. Il ne peut y avoir aucun élément de ponctuation ni d'espace dans un nom de variable, sinon, la plupart du temps, vous aurez une erreur de type `syntaxError` :

```

1 chou-fleur = 42
2 SyntaxError: can't assign to operator

```

Par ailleurs, une variable ne peut commencer par un chiffre :

```

1 2020Tokyo = 32
2 SyntaxError: invalid syntax

```

d) Nom ou variable non reconnus

Une variable qui est accédée sans avoir été affectée au préalable n'est pas reconnue :

```

1 R = 10
2 print(2 * pi * R)
3 NameError: name 'pi' is not defined

```

— Ligne 2 : la variable `pi` est utilisée mais n'a jamais été défini auparavant.

— Ligne 3 : le nom de variable `pi` n'est pas reconnu d'où le message `NameError` qui doit s'interpréter par : « nom inconnu ».

Ce type d'erreur provient parfois d'une erreur de saisie du code (une « coquille ») :

```

1 toto = 42
2 print(totot * 10)
3 NameError: name 'totot' is not defined

```

Ce phénomène s'applique aussi à des noms (par exemple, de fonctions « officielles » du langage), pas seulement des variables :

```

1 toto = 42
2 printf(toto * 10)
3 NameError: name 'printf' is not defined

```

— Ligne 2 : le programmeur a saisi `printf` au lieu de `print`.

e) Réaffectation de variable

Lorsqu'un calcul avec une variable `toto` est suivie de la modification de la valeur de la variable `toto`, on dit que `toto` a subi une réaffectation :

```

1 x = 42
2 x = 2 * x + 10
3 print(x)
4 94

```

- Ligne 2 : l'expression `2 * x + 10` est évaluée et la valeur obtenue (94) est réaffectée à `x`. Ce type d'affectation (ligne 1 vs ligne 2) s'appelle une *réaffectation*. La valeur initiale d'affectation (ici 42) est « perdue ».

Augmenter de 1 la valeur d'une variable (comme `toto` ci-dessous) s'appelle une *incrément* :

`incrementation.py`

```

1 toto = 42
2 print(toto)
3
4 toto = toto + 1
5 print(toto)
6 42
7 43

```

- Ligne 4 : l'expression `toto + 1` est évaluée et la valeur obtenue (43) est réaffectée à `toto`.

La nécessité d'effectuer une réaffectation est extrêmement courante.

f) Affichage multiple

Il est possible d'afficher plusieurs objets sur une même ligne à l'aide d'un seul appel à `print`.

```

1 print("toto", 42+100)
2 toto 142

```

- Ligne 1 : on affiche la chaîne `toto` et le nombre `42+100`.
- Ligne 2 : les deux objets à afficher (`toto` et le nombre 152) sont séparés exactement d'un seul espace.

Les objets sont affichés sur une même ligne, séparés par un seul espace. Placer plus ou moins d'espacements dans l'écriture de l'appel à la fonction `print` ne modifie nullement l'espacement affiché :

```

1 print("toto" , 42+100)
2 toto 142

```

- Ligne 1 : lors de l'écriture de l'appel dans le code-source, de nombreux espaces ont été placés entre les deux objets à afficher.
- Ligne 2 : l'espacement à l'affichage reste pourtant d'une espace.

La possibilité d'affichage multiple s'applique aussi à des expressions dépendant de variables :

```

1 x = 42
2 y = 10
3 print(x + y, x - y)
4 52 32

```

5 Constantes, nombres

a) Notions sur les constantes

Soit le code Python suivant :

```
1 z = 31
2 a = 42 * z + 81
3 z = "toto"
4 pi = 3.14159
```

Dans ce code,

31, 42, 81, "toto" ou 3.14159

sont des *constantes* du langage Python, Elles sont dites parfois *littérales* car leur façon de s'écrire dans le code permet de connaître immédiatement leur valeur, sans faire aucune opération. Par contre,

42 * z + 81

N'est PAS une constante (c'est juste une *expression* qui utilise des constantes). Les constantes sont des briques élémentaires (parmi d'autres) des « écritures » d'un programme.

Pour afficher une constante présente dans un code Python, on utilise la fonction **print**.

```
1 a=42
2 print(a)
3
4 z="toto"
5 print(z)
6 print("toto")
7
8 pi = 3.14159
9 print(pi)
```

```
10 42
11 toto
12 toto
13 3.14159
```

- Lignes 5 ou 9 : instruction d'affichage du contenu d'une variable dont le contenu (lignes 4 ou 8) est une constante.
- Ligne 6 : on peut aussi obtenir l'affichage de la constante sans l'avoir placée dans une variable.
- Lignes 10-13 : les affichages correspondants.

b) Constantes entières

Les constantes entières figurant textuellement dans un code , telles que 421, peuvent être précédées d'un signe - pour désigner un entier négatif ou d'un signe +, facultatif et en général omis :

```
1 print(-42)
2 print(+421)
```

```
3 -42
4 421
```

Toute constante entière, écrite de la façon habituelle avec des chiffres entre 0 et 9, comme 421 ou 0 est interprétée par défaut comme écrite en base 10.

Attention à ne pas placer involontairement un 0 comme premier chiffre d'une constante entière, comme on le ferait pour certaines dates (par exemple 01 dans la date **19/01/2038**) sinon on obtient un message d'erreur :

```
1 print(042)
```

```
2 File "_py", line 1
3     print(042)
4         ^
5 SyntaxError: invalid token
```

c) Grands entiers

On peut effectuer tout type d'opération arithmétique avec des entiers aussi grands qu'on le souhaite. Une opération arithmétique ne peut jamais créer d'« overflow » comme c'est le cas en C ou Java où les entiers sont codés sur un nombre prédéfini de bits.

Exemple d'usage de grands entiers :

```
1 a = 8596125023601452569859
2 b = 7012365895410232252363
3 print(a * b)
4 60279173948185303803365326980576772175326817
```

d) Constantes flottantes

La présence d'un point dans une constante numérique donne à la constante le type flottant. Le point est, dans ce cas, l'équivalent de notre virgule décimale :

```
1 print(3.14)
```

```
2 3.14
```

— Ligne 1 : le nombre 3.14 est une constante flottante ; il y a des chiffres de part et d'autre du point

— Ligne 2 : l'affichage est obtenu via la fonction **print**.

Il est parfois possible qu'aucun chiffre ne précède ou ne suit le point :

```
1 print(1.)
2 print(.25)
```

```
3 1.0
4 0.25
```

— Ligne 1 : 1. est équivalent à 1.0

— Ligne 2 : .25 est équivalent à 0.25

Le but de ce type d'écriture est juste d'alléger la saisie du code.

En France, le séparateur décimal est une virgule. Il faut cependant toujours utiliser un point pour désigner le séparateur d'une constante flottante. Voici un exemple de comportement inattendu :

```
1 pi = 3,14
2 print(pi + 1)
```

```
3 TypeError: can only concatenate tuple (not "int") to tuple
```

— Ligne 1 : on déclare le nombre `pi` sous la forme `3,14` au lieu de `3.14`.

— Lignes 2 : On veut ajouter 1 à `pi` (on devrait obtenir `4,14`) mais le programme est interrompu.

On peut faire précéder une constante flottante d'un signe `-` ou `+` :

```
1 print(-3.14)
2 print(+3.14)
```

```
3 -3.14
```

```
4 3.14
```

6 Les types

a) Types de base

Les objets manipulés par un programme Python ont un *type*, au sens d'un langage de programmation : un type correspond à une catégorie d'objets (entiers, nombres réels, valeurs logiques, etc).

Voici quelques exemples de types usuels :

```
1 print(42)
2 print(4 + 6)
3 print(-5 * 2)
4
5 print(421.5)
6 print(2.7 + 10)
7
8 print(421 > 42)
9 print(421 < 42)
10
11 print("toto")
```

```
12 42
13 10
14 -10
15 421.5
16 12.7
17 True
18 False
19 toto
```

— Lignes 1-3 : type *entier*. Les entiers peuvent être positifs comme négatifs.

- Lignes 5-6 : type *flottant*. Pour simplifier, il s'agit de nombres dit « à virgule », ci-dessus représentés en base 10. Le point utilisé dans le nombre représente notre virgule décimale (« flottant » fait allusion à la notion de virgule « flottante »).
- Lignes 8-9 et 17-18 : type *booléen*. Une expression de type booléen a une valeur de vérité **True** ou **False**.
- Lignes 11 : le type *chaîne*. En première approximation, une chaîne représente une suite de caractères. Dans l'exemple, pour que Python reconnaisse le mot **toto** comme une donnée de type chaîne, on entoure le mot d'une paire de guillemets.

b) Variables et typage

Un nom de variable n'est pas associé à un type fixé

```
1 a = 2020
2 print(a)
3 a = 3.14
4 print(a)
```

```
5 2020
```

```
6 3.14
```

- Ligne 1 : l'étiquette **a** se réfère d'abord à un entier.
- Ligne 3 : l'étiquette **a** se réfère ensuite à un flottant.

Une variable n'a pas de type. Ce qui a un type est l'objet placé en mémoire et que la variable référence.

Une variable peut étiqueter un objet de n'importe quel type.

Par exemple, on peut placer une variable sur le nom d'une fonction :

```
1 p = print
2 p("bonjour")
```

```
3 "bonjour"
```

c) Les fonctions built-in

Python propose, par défaut, des fonctions prêtes à l'emploi (les fonctions dites « built-in »⁶) et qui permettent de faire des opérations courantes en programmation : afficher, trier une liste, trouver le plus grand nombre d'une liste, calculer la longueur d'une chaîne de caractères, etc.

Exemple

```
1 print(max(81, 12, 31, 82, 65))
```

```
2 82
```

- Ligne 1 : **max** est une fonction built-in. Elle renvoie le plus grand élément des nombres qu'on lui transmet.

La fonction built-in la plus utilisée est probablement la fonction **print**.

6. Que l'on pourrait traduire par *fonctions intégrées*.

7 Opérations

a) Opérations sur les nombres

Voici quelques opérations usuelles que l'on peut effectuer entre nombres réels (pas seulement entiers).

On peut calculer des expressions mathématiques avec des parenthèses :

```
1 print(-421)
2 print(2 * 10 + 15)
3 print((2 + 3) * (8 + 2))
4 print(42.1 / 10)
```

```
5 -421
6 35
7 50
8 4.21
```

— Lignes 4 et 8 : on effectue la division « exacte » de 42.1 par 10.

Certains calculs nécessitent le placement de parenthèses. Par exemple, en mathématiques, on écrit couramment

$$\frac{a}{bc}.$$

En Python, cela NE se traduit PAS par `a / b * c` mais par `a / (b * c)` :

```
1 x = 100 / 10 * 5
2 y = 100 / (10 * 5)
3 print(x, y)
```

```
4 50.0 2.0
```

En cas de doute dans un calcul, placer des parenthèses même s'il est possible qu'elles ne soient pas indispensables. Ainsi, la fraction

$$\frac{\frac{100.0}{5}}{4}$$

sera traduite par `(100. / 5) / 4` :

```
1 x = (100 / 5) / 4
2 y = 100 / 5 / 4
3 print(x, y)
```

```
4 5.0 5.0
```

Ne pas utiliser de crochets à la place des parenthèses car les crochets ont un autre sens en Python.

Pour calculer une puissance x^y , on écrit `x ** y` dans le code Python :

```
1 print(10 ** 2)
2 print(6.25 ** 0.5)
3 print(0 ** 0)
```



```
4 100
5 2.5
6 1
```

Les nombres x et y peuvent n'être pas des entiers.

En particulier, `x ** 0.5` représente \sqrt{x} , la racine carrée de x (lignes 2 et 5)

Les puissances de 10

Python permet en particulier de manipuler aisément les puissances de 10.

```
1 print(1000000000000000)
2 print(10**13)
3 1000000000000000
4 1000000000000000
```

— Ligne 1 : écriture, relecture délicates

— Ligne 2 : écriture, relecture immédiates

L'utilisation de puissances de 10 permet d'éviter des difficultés de saisie et de lecture dues à un grand nombre de zéros. Par exemple, pour écrire

— deux millions, utiliser la notation `2 * 10 ** 6`

— deux milliards, utiliser la notation `2 * 10 ** 9`

b) Utiliser des variables

Quand on veut « récupérer » (autrement dit *sauvegarder*) le résultat d'un calcul, on le place dans une nouvelle variable :

```
1 x = 42
2 y = 5
3 z = (x * y + 5) * (x ** 2 + y **2 ) - 100
4 u = (z - 10000) * x
5
6 print(u)
7 15730470
```

— Lignes 3 : on stocke sous le nom de `z` le résultat d'un calcul en mémoire

— Lignes 4 : on stocke sous le nom de `u` le résultat d'un autre calcul

— Ligne 6 : on affiche le contenu de la variable

Une expression utilisant une variable ne modifie pas le contenu de la variable :

```
1 toto = 42
2 print(toto + 1)
3 print(toto)
4 43
5 42
```

— Ligne 2 : on ajoute 1 à la **valeur** de `toto`. Le résultat (ici 43) est affichée mais ce calcul est « perdu » puisqu'il n'est pas placé dans une variable.

— Ligne 5 : bien qu'on ait ajouté 1 à `toto`, le contenu de la variable `toto` reste inchangé par rapport à sa valeur initiale.

c) Division flottante

Pour obtenir la division *exacte*⁷ du nombre `a` par le nombre `b`, on utilise tout simplement la syntaxe `a / b` :

```
1 print(366/ 12)
2 print(3.14 / 2)
3 30.5
4 1.57
```

Les nombres `a` et `b` peuvent aussi bien être des entiers que des flottants. Le résultat `a / b` sera toujours de type flottant.

d) Opérations mixtes entre flottants et entiers

Lorsqu'on effectue un ensemble d'opérations arithmétiques qui font intervenir des entiers et des flottants, le résultat est de type flottant :

```
1 x = 1.0
2 y = 42
3
4 print(x * y)
5 print(x - x)
6 42.0
7 0.0
```

— Lignes 4 et 5 : bien que les valeurs des flottants représentent des nombres entiers, le résultat est de type flottant

e) Opérateurs « égale à » et « différent de »

Opérateur ==

On peut tester l'égalité de *valeur* de deux objets avec l'opérateur `==` :

```
1 print(42 == 21 + 21)
2 print(42 == 10 + 10)
3 True
4 False
```

— Lignes 3 : si les valeurs sont égales l'opérateur `==` ligne 1 renvoie la valeur logique **True**

— Ligne 4 : si les valeurs comparées ne sont pas égales (42 et 20) l'opérateur `==` renvoie **False**.

Il faut effectivement écrire deux fois de suite le signe `=`, sans espace entre les deux.

7. Le terme de division *exacte* est abusif puisque la division `a / b` n'a aucune raison d'être *exacte* au sens mathématique du terme mais elle ne sera bien souvent qu'une approximation. Par exemple, $1/3$ n'est pas *strictement* égal à 0.3333333333333333.

Opérateur !=

Pour tester si deux objets ont des valeurs distinctes, on utilise l'opérateur != (lire : *différent de*). C'est l'opérateur « contraire » de == :

```
1 print(42 != 21 + 21)
2 print(42 != 10 + 10)
3 False
4 True
```

f) Comparaison de nombres

Le langage Python dispose d'opérateurs permettant d'effectuer des comparaisons de nombres :

```
1 print(42 > 24)
2 print(42 <= 24)
3 print(3.14159 < 2.71828)
4 print(24 >= 2 * 12)
5 True
6 False
7 False
8 True
```

On peut comparer des nombres au sens des opérateurs mathématiques suivants :

$< \leq > \geq$

En Python, on utilise les opérateurs suivants :

$<, <=, > \text{ et } >=$

Une comparaison renvoie un booléen **True** ou **False** (*Vrai* ou *Faux*) qui précise si l'inégalité écrite est vraie ou fausse.

Encadrement

On peut utiliser les opérateurs de comparaison en les enchaînant, comme on le fait usuellement en mathématiques :

```
1 print(42 < 81 < 100 <= 512)
2 print(42 < 81 < 100 < 100)
3 True
4 False
```

g) Comparaison de flottants

L'arithmétique en nombres flottants est approchée. L'exemple ci-dessous illustre le type de problème rencontré :

```
1 print(6 * 0.7 == 4.2)
2 False
```

— Ligne 1 : mathématiquement, $6 \times 0,7 = 4,2$

— Ligne 2 : Python ne considère pas les deux nombres comme identiques.

D'où la conséquence suivante :

Règle fondamentale : on n'utilise *jamais* l'opérateur `==` pour comparer deux flottants.

h) Confusion entre égalité et affectation

Le signe `=` est le signe de l'affectation. Il ne faut pas le confondre avec l'opérateur `==` d'égalité de valeurs. C'est une source fréquente d'erreur d'inattention, y compris chez les programmeurs non débutants.

Assez souvent, la confusion déclenche un message d'erreur :

```
1 x == 42
2 print(x)
3 Traceback (most recent call last):
4   File "_py", line 1, in <module>
5     x == 42
6 NameError: name 'x' is not defined
```

— Ligne 1 : confusion de `==` avec `=` : la ligne examine si les valeurs de `x` et 42 sont égales mais `x` n'a pas été défini.

— Ligne 6 : la nature de l'erreur est confirmée par le message.

Autre erreur possible :

```
1 x = 42
2 print(x = 421)
3 Traceback (most recent call last):
4   File "egalite_vs_affectation.py", line 2, in <module>
5     print(x = 421)
6 TypeError: 'x' is an invalid keyword argument for this function
```

— Ligne 2 : confusion de `=` avec `==` (une affectation n'est pas une expression donc on ne peut pas l'afficher)

— Lignes 3-6 : l'erreur est plus difficile à interpréter et nécessite des connaissances plus poussées sur la fonction `print` et les fonctions en général.

Mais parfois, il n'y a aucun déclenchement d'erreur :

```
1 x = 42
2 print(x)
3 x == 100
4 print(x)
5 42
6 42
```

— Ligne 3 : sans doute le programmeur voulait-il écrire une nouvelle affectation `x = 100`. Mais cette erreur de programmation ne déclenche aucune erreur d'exécution.

8 Division entière

a) Multiples, division entière

On rappelle quelques notions d'arithmétique élémentaire.

Multiples

Les nombres qui sont les résultats dans une table de multiplication sont appelés des *multiples*. Par exemple, les 10 premiers multiples de 9 sont :

1	1 x 9 = 9
2	2 x 9 = 18
3	3 x 9 = 27
4	4 x 9 = 36
5	5 x 9 = 45
6	6 x 9 = 54
7	7 x 9 = 63
8	8 x 9 = 72
9	9 x 9 = 81
10	10 x 9 = 90

Plus généralement, tout multiple de 9 est de la forme $9 \times k$ où k est un entier. Par exemple, $720 = 9 \times 80$ est un multiple de 9.

Si un nombre n est un multiple de 9, il est équivalent de dire que 9 est un *diviseur* de n ou encore que n est *divisible* par 9.

Réciproquement, soit le problème suivant :

474 est-il un multiple de 8 ?

Pour répondre, il suffit de faire la division à la façon de « l'école primaire » de 474 par 8 :

$$\begin{array}{r|l} 474 & 8 \\ \hline 74 & 59 \\ 2 & \\ \hline \end{array}$$

↑
Reste

↑
Quotient entier

La division de l'école primaire

Si le reste de cette division vaut 0 autrement dit, si la division tombe juste, c'est que la réponse est *oui* et si le reste est non nul, la réponse est *non*. Ici, la division de 474 par 8 admet 2 pour reste, en sorte que la réponse est *non*.

La division de l'école primaire est en fait appelée *division entière*. La division entière fournit un *reste* et un *quotient*. Par exemple, la division entière de 474 par 8 admet 59 pour quotient et 2 pour reste.

Plus généralement, quand on effectue la division entière de a par b , on obtient un quotient q et un reste r . Noter que r vérifie : $0 \leq r < b$ autrement dit le reste est toujours plus petit que le diviseur.

La division entière est différente de la division dite « exacte ». La division exacte ne fournit qu'un quotient, supposé exact. Par exemple, la division exacte de 474 par 8 est 59.25, ce qui s'écrit

$$\frac{474}{8} = 59.25$$

En réalité, très souvent, le quotient obtenu est une valeur approchée : par exemple, on écrit couramment $\frac{474}{7} = 67.71$ bien qu'en fait 67.71 ne soit qu'une approximation de la division de 474 par 7, qui « ne tombe pas juste ».

La notion de multiple donne fréquemment lieu à des questions de programmation.

b) Quotient et reste de division entière

Le quotient **entier** de l'entier a par l'entier b est obtenu par l'opération $a // b$ (double oblique) :

```
1 print(42 // 10)
2 4
```

— Ligne 1 : Noter le double slash pour effectuer la division entière, et non $42 / 10$.

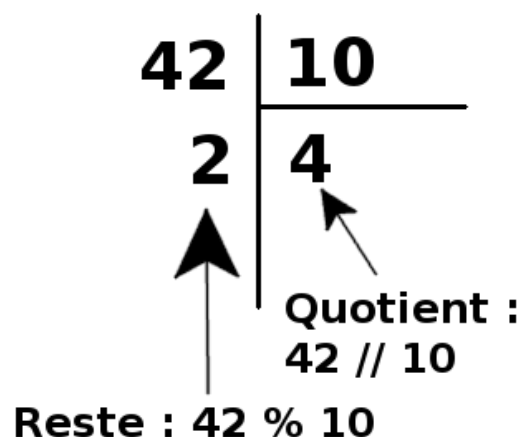
— Ligne 2 : noter que la réponse n'est pas **4,2** (il ne s'agit pas de la division exacte).

Le reste entier de la division de l'entier a par l'entier b est noté $a \% b$:

```
1 print(42 \% 10)
2 2
```

— Ligne 1 : le calcul du reste se fait avec l'opérateur $\%$ « pourcent ».

L'opérateur $\%$ est parfois appelé *opérateur modulo*.



Division par zéro

Toute forme de division par 0 est interdite et renvoie une erreur :

```
1 print(42 // 0)
2 Traceback (most recent call last):
3   File "division_par_zero.py", line 1, in <module>
4     print(42 // 0)
5 ZeroDivisionError: division by zero
```

— Ligne 1 : quotient par 0 : interdit

De même :

```
1 print(42 % 0)
2 Traceback (most recent call last):
3   File "reste_par_zero.py", line 1, in <module>
4     print(42 % 0)
5 ZeroDivisionError: integer division or modulo by zero
```

— Ligne 1 : reste en divisant par 0 : interdit

c) Division entière et quotient exact

Si b est un entier multiple de l'entier d , la division exacte b/d a même valeur (mathématique) que le quotient entier de b par d . Mais en Python, le quotient exact sera de type flottant tandis que le quotient entier de type entier :

```
1 b = 42
2 d = 6
3
4 print(b // d)
5 print(b % d)
6
7 print(b / d)
8 7
9 0
10 7.0
```

— Lignes 5 et 9 : b est bien un multiple de d puisque le reste est nul.

— Lignes 8 et 10 : le quotient entier et le quotient exact ont même valeur mathématique mais pas même type.

Bien que les valeurs des quotients soient exactes, la différence de type peut entraîner des erreurs. Par exemple, la fonction `range` n'admet en arguments que des objets de type entier :

```

1 b = 42
2 d = 6
3
4 print(b // d)
5 print(b % d)
6
7 print(b / d)
8
9
10 print(list(range(b // d)))
11 print("-----")
12
13
14 # Va entraîner une erreur
15 print(list(range(b / d)))

```

```

16 7
17 0
18 7.0
19 [0, 1, 2, 3, 4, 5, 6]
20 -----
21 Traceback (most recent call last):
22   File "division_entiere_et_quotient_exact.py", line 13, in <module>
23     print(list(range(b / d)))
24 TypeError: 'float' object cannot be interpreted as an integer

```

Il se peut même que les quotients calculés soient différents alors que les valeurs mathématiques soient égales :

```

1 b = 1874150539755119646735
2 d = 208065
3 print(b // d)
4 print(b % d)
5
6 print(b / d)

```

```

7 9007524282099919
8 0
9 9007524282099920.0

```

En conclusion, si on cherche le quotient exact d'un entier b par un de ses diviseurs, on utilisera l'opérateur de division entière.

9 Plus d'affichage

a) Affichage amélioré

Le programmeur a parfois besoin de rendre plus lisible des résultats à afficher dans un terminal ou une zone d'output. Voici un exemple sommaire d'une telle situation :

`afficherLisible.py`

```
1 a = 100
2 b = 42
3 print(a + b, a - b)
4 print("L'addition -> ", a + b, "La soustraction -> ", a - b)

5 142 58
6 L'addition -> 142 La soustraction -> 58
```

— Ligne 3 et ligne 5 : affichage peu lisible.

— Ligne 4 et ligne 6 : affichage dont le contenu est plus facilement interprétable.

Affichage et débogage

En particulier, pour vérifier (visuellement) la justesse d'un programme ou pour mieux en comprendre le fonctionnement, il est parfois utile de faire des affichages des variables et des valeurs calculées :

```
1 a = 100
2 b = 42
3 print("a = ", a, ", ", "b = ", b)
4 print("-----")
5 print("a + b = ", a + b)
6 print("a - b = ", a - b)

7 a = 100 , b = 42
8 -----
9 a + b = 142
10 a - b = 58
```

L'action de rechercher les erreurs (les « bugs » ou, en français, les « bogues ») que contient un programme s'appelle le *débogage*. Le placement d'instructions d'affichage dans certaines parties bien choisies d'un code-source en cours de réalisation dans le but de mieux comprendre comment il s'exécute et ce que valent les variables pendant l'exécution du code est une forme très fréquente de débogage.

b) Afficher un message complexe

Comme un texte est une suite de lignes, la fonction `print` permet d'afficher des messages plus élaborés :

```
1 print("Voici les couleurs")
2 print()
3 print("+-----+-----+-----+")
4 print("| Rouge | Vert | Bleu |")
5 print("+-----+-----+-----+")
```

qui affiche

```
1 Voici les couleurs
2
3 +-----+-----+-----+
4 | Rouge | Vert | Bleu |
5 +-----+-----+-----+
```

Noter, qu'en général, ce type de message est compliqué à construire et à maintenir.

c) Affichage et forcer un passage à la ligne

Pour séparer deux affichages bien distincts d'un même programme, le programmeur veut parfois placer un ou plusieurs sauts de ligne entre les parties à afficher. Par exemple, supposons qu'il veuille afficher

```
1 un
2 deux
3 trois
4
5
6 1
7 2
8 3
```

Il est sans effet de passer des sauts de lignes dans le code-source. Ainsi, le code suivant

```
1 print("un")
2 print("deux")
3 print("trois")
4
5
6 print(1)
7 print(2)
8 print(3)
```

affichera

```
1 un
2 deux
3 trois
4 1
5 2
6 3
```

(noter les sauts de lignes dans le code-source aux lignes 4-5).

Une solution au problème est la suivante :

```
1 print("un")
2 print("deux")
3 print("trois")
4 print()
5 print()
6 print(1)
7 print(2)
8 print(3)
```

```
9 un
10 deux
11 trois
12
13
14 1
15 2
16 3
```

- Lignes 4-5 : appel à la fonction `print` sans argument (une paire de parenthèses vides).
- Lignes 12-13 : chaque appel `print()` effectue un saut de ligne à l’affichage ce qui « allège » l’affichage final.

10 Saisie du code

a) Indentation accidentelle

Le placement d’espaces horizontaux **en début** de ligne n’est autorisé que dans certaines circonstances très strictes.

En cas de non respect de ces règles, on obtient une interruption du programme :

```

1  x = 3000 - (2038 - 38)
2  print("Bonjour !", x)

3  File "main.py", line 1
4      x = 3000 - (2038 - 38)
5      ^
6  IndentationError: unexpected indent

```

Le message `IndentationError` signifie que des espaces en **début** de ligne sont mal placées.

b) Placer des commentaires

Il est possible d’annoter son code-source par des commentaires personnels. Voici un exemple de code-source commenté :

```

1  # Mon joli code
2
3
4  # Voici un exemple
5  # d'addition
6  a=4+2
7
8      # on va afficher le résultat
9  print(a) # affichage
10 # affiche 6

```

Un commentaire Python est introduit par un caractère dièse⁸ comme ceci : `#` (cf. ligne 4 par exemple).

Pour que le caractère `#` ait une valeur de commentaire, aucun caractère autres que d’éventuels espaces ne peuvent le précéder dans la ligne où le caractère `#` est placé. Le code `commentaire.py` contient 6 commentaires.

La présence de commentaires n’a pas d’impact sur l’exécution du code⁹ et ils sont supprimés du code avant l’exécution de celui-ci. Le code ci-dessus est exactement équivalent au code suivant :

`commentaires_repires.py`

```

1  a=4+2
2  print(a)

```

8. Le caractère porte différents noms : dièse, sharp, « pound », « hash » (ce dernier terme est utilisé dans la documentation officielle) ; selon Wikipedia, le terme français correct semblerait être « croisillon ».

9. Un commentaire n’est pas une instruction du langage.

Les commentaires sont destinés au programmeur lui-même ou à un autre programmeur qui va lire le code afin d'en expliquer certaines parties.

11 Compléments

a) L'affectation

L'affectation permet à une variable de référencer un objet placé en mémoire.

```
1 a = "azertyuiopmlkjhgfdsqwxcvbn"
2 print(a)
3 a = 42
4 print(a)
```

```
5 azertyuiopmlkjhgfdsqwxcvbn
6 42
```

— Ligne 1 : la variable `a` référence la liste des touches d'un clavier AZERTY

— Ligne 3 : la variable `a` est réaffectée et référence une autre donnée. La chaîne `"azertyuiopmlkjhgfdsqwxcvbn"` est désormais perdue, on ne peut plus y accéder.

Voici un autre code :

```
1 a = "azertyuiopmlkjhgfdsqwxcvbn"
2 print(a)
3 b = a
4 a = 42
5 print(a)
6 print(b)
```

```
7 azertyuiopmlkjhgfdsqwxcvbn
8 42
9 azertyuiopmlkjhgfdsqwxcvbn
```

— Ligne 1 : la variable `a` référence la liste des touches d'un clavier AZERTY.

— Ligne 3 : la variable `b` est une deuxième référence vers la liste des touches d'un clavier AZERTY

— Ligne 4 : la variable `a` est réaffectée et référence une autre donnée.

— Lignes 6 et 9 : La chaîne `"azertyuiopmlkjhgfdsqwxcvbn"` n'est pas pour autant perdue puisqu'elle est encore référencée par la variable `b`.

Alias

Si `a` est une variable référençant un objet `v` et `b` la variable définie par `b = a` alors on dit que `b` est un alias de `a`. La création d'un alias ne crée jamais de copie de `v` mais seulement une nouvelle **référence** vers `v`.

Affectation versus égalité mathématique

Malgré l'usage du signe `=`, l'affectation ne doit pas être confondue avec l'égalité mathématique. Par exemple, en mathématiques, les relations $a = 5$ et $5 = a$ sont équivalentes. Il n'est pas de même en Python :

```

1 a = 5
2 print(a)
3 5

```

tandis que le code suivant provoque un message d'erreur

```

1 5 = a
2 print(a)
3 File "<ipython-input-35-2af213e781d3>", line 1
4 SyntaxError: can't assign to literal

```

De même, en mathématiques, les égalités $a = 0$, $b = a$ et $b = 42$ entraînent $a = b = 42$ ce qui n'est pas le cas en Python :

```

1 a=0
2 b=a
3 b=42
4 print(a)
5 print(b)
6 0
7 42

```

b) Affectation combinée

Une opération suivie d'une réaffectation comme

```

1 x = 42
2 print(x)
3 x = x + 100
4 print(x)
5 42
6 142

```

est une opération courante.

Il existe donc un raccourci syntaxique pour ce type de réaffectation de variable :

```

1 x = 42
2 print(x)
3 x += 100
4 print(x)
5 42
6 142

```

Dans ce contexte où les variables réfèrent des nombres, la syntaxe `x += a` est équivalente à `x = x + a`. On dit que `+=` est une affectation combinée¹⁰. Rajouter un nombre à une variable s'appelle une *incrémement*. Une incrémentation de l'entier 1 est un cas particulier fréquent :

10. *augmented assignment* en anglais

```

1 x = 42
2 print(x)
3 x += 1
4 print(x)

```

```

5 42
6 43

```

Il n'existe pas de syntaxe spécifique pour traduire `x += 1` par `x++` comme cela se fait en C ou Java.

La syntaxe appliquée pour `+` existe pour d'autres opérateurs, par exemple, `-`, `*`, `/`, `//` et `%` (liste non exhaustive). Ainsi :

```

1 x = 42
2 print(x)
3 x -= 10
4 print(x)

```

```

5 42
6 32

```

D'une façon générale, si `x` est une variable et si `TRUC` est un opérateur comme `+`, alors le code

$$x \text{ TRUC} = a$$

a une sémantique équivalente à

$$x = x \text{ TRUC } a.$$

Toutefois, dans certaines circonstances, `x TRUC = a` et `x = x TRUC a` peuvent avoir des exécutions non *strictement* équivalentes.

c) Partie entière et type `int`

On rappelle que la partie entière de, par exemple, $x = 4.21$ est 4 et la partie entière de $x = -4.21$ est -5 . Mathématiquement, la partie entière de x est le plus grand entier n tel que $n \leq x$.

Le type `int` permet de déterminer la partie entière d'un nombre flottant positif :

```

1 x = int(4.21)
2 print(x)

```

```

3 4

```

On notera que `int(4.21)` renvoie un entier et non le flottant `4.0`.

Pour les nombres négatifs, `int` ne renvoie pas la partie entière au sens mathématiques du terme :

```

1 x = int(-4.21)
2 print(x)

```

```

3 -4

```

— La partie entière de $-4,21$ est -5 et non -4 .

Donc, le type `int` appliqué à un flottant retire juste la partie décimale du flottant.

d) La bibliothèque standard

Le langage Python intègre la possibilité, par exemple, de trier une liste d'entiers :

```
1 print(sorted([5,19,14,12,21]))  
2 [5, 12, 14, 19, 21]
```

— Ligne 1 : la liste 5,19,14,12,21 n'est pas triée.

— Ligne 2 : la liste [5, 12, 14, 19, 21] est triée.

À l'inverse de la fonction de tri `sorted`, certaines fonctions, comme la partie entière d'un nombre, ne peuvent être utilisées *directement* par le langage Python mais sont disponibles si on fait appel à ce qu'on appelle la « bibliothèque standard » de Python.

Ainsi, voici comment on peut calculer en Python des parties entières :

```
1 from math import floor  
2  
3 print(floor(4.21))  
4 print(floor(-4.21))  
5 4  
6 -5
```

— Ligne 1 : on doit importer la fonction `floor` du « module » `math` de la bibliothèque standard.

— Lignes 3 et 4 : on appelle la fonction de partie entière `floor` importée du module `math`.

La notion de bibliothèque standard

La bibliothèque standard de Python est livrée avec le langage Python mais ne fait pas partie du langage en tant que tel. Elle *étend* les possibilités du langage.

Les fonctionnalités de la bibliothèque standard sont regroupées en modules. Ainsi, les fonctionnalités mathématiques sont disponibles dans le module `math`.

Pour accéder depuis du code Python à une fonctionnalité d'un module de la bibliothèque standard, il faut importer cette fonctionnalité dans le code Python par une instruction d'importation (cf. ligne 1 ci-dessus).

La bibliothèque standard contient beaucoup d'autres modules que le module `math`. Voici quelques exemples de modules disponibles :

<code>random</code>	Manipuler des nombres aléatoires
<code>decimal</code>	Résultats exacts avec les nombres décimaux
<code>datetime</code>	Gestion des dates, des durées
<code>os.path</code> , <code>shutil</code>	Manipuler le système de fichiers
<code>Tkinter</code>	Interfaces graphiques

Un module de la bibliothèque standard est dit *module standard*.

De nombreuses fonctionnalités de la bibliothèque standard concernent des questions spécialisées et/ou techniques et sont réservés à des programmeurs de niveau avancé voire professionnel.

e) Importer une fonctionnalité standard

Soit le code suivant qui importe la fonction `floor` du module standard `math`.

```

1 from math import floor
2
3 print(floor(4.21))

```

4 4

Ligne 1 : on n'importe qu'une seule fonctionnalité, la fonction `floor`.

Pour qu'un code Python puisse utiliser une fonctionnalité, par exemple la fonction `floor`, ce code doit importer (avec une instruction `import`) la fonctionnalité du module ad hoc.

Si on n'importait rien, voilà ce que cela produirait :

```

1 z = floor(421.2020)
2 print(z)

```

```

3 ----> 1 z = floor(421.2020)
4         2 print(z)
5
6 NameError: name 'floor' is not defined

```

— Ligne XX : Faute d'importation, le nom `floor` n'est pas reconnu.

On a importé une *fonction* du module `math` mais un module peut, en général, proposer d'autres types d'objets à importer, par exemple des constantes. Ainsi, le module `math` dispose de la constante mathématique π :

```

1 from math import pi
2
3 print(pi)
4 3.141592653589793

```

EXERCICES TYPE

Volume de la sphère

Le volume de la sphère est $V = \frac{4}{3}\pi r^3$. Écrire en Python une formule (avec des variables) qui calcule le volume V d'une sphère de rayon r . On posera `pi = 3.14`. Tester avec $r = 10$.

Solution

D'abord, écrire la formule de manière purement formelle ne fonctionne pas :

```

1 v = 4/3 * pi * r**3
2 print(v)

```

```

3 NameError                                Traceback (most recent call last)
4 <ipython-input-32-b6f8fcc9c76e> in <module>()
5 ----> 1 v = 4/3 * pi * r**3
6         2 print(v)
7
8 NameError: name 'pi' is not defined

```

Python n'est pas un outil de calcul formel.

Il faut initialiser r mais aussi π :

```
1 pi = 3.14
2
3 r=10
4 v = 4/3 * pi * r**3
5 print(v)
6 4186.666666666667
```

Pour tester d'autres valeurs le plus simple est de changer r dans le code-source et de relancer l'exécution du code.

Écart entre deux heures de la journée

On donne deux moments de la journée a et b , par exemple $a = 8\text{h } 13\text{min } 17\text{s}$ et $b = 19\text{h } 24\text{min}$ et on demande de donner en heures, minutes, secondes le temps qui s'est écoulé entre les deux moments.

Chaque moment de la journée sera codé par trois variables représentant les heures, les minutes et les secondes.

On pourra utiliser le canevas de code suivant :

```
1 h1=8
2 m1=13
3 s1=17
4 h2=19
5 m2=24
6 s2=0
7
8 # votre code CI-DESSOUS
```

Dans l'exemple précédent, votre code doit afficher 11 h 10 min 43 s.

Solution

Pour calculer l'écart entre deux moments de la journée, il suffit

- de convertir chaque moment en secondes
- d'effectuer la différence des moment en secondes
- de convertir cet écart en heures, minutes et secondes.

D'où le code suivant :

```

1 h1=8
2 m1=13
3 s1=17
4 h2=19
5 m2=24
6 s2=0
7
8 moment1= h1*3600+m1*60+s1*1
9 moment2= h2*3600+m2*60+s2*1
10
11 ecart=moment2-moment1
12
13 h=ecart//3600
14 r=ecart%3600
15 m=r//60
16 s=r%60
17 print(h,"h ", m, "min ", r%60, "s")
18 11 h 10 min 43 s

```

- Lignes 8-9 : chaque moment est converti en secondes
- Ligne 11 : l'écart en secondes est calculé.
- Lignes 12-16 : conversion de la durée en secondes au format initial, c'est-à-dire en heures, minutes et secondes.

EXERCICES

Echanger deux variables

Étant donné deux variables x et y , on veut permuter leurs valeurs. Par exemple si au départ $x = 81$ et $y = 31$ alors, après échange, on doit avoir : $x = 31$ et $y = 81$. On pourra compléter le code ci-dessous :

```

1 x=81
2 y=31
3 print(x,y)
4
5 # VOTRE
6 # CODE
7 # ICI
8
9 print(x,y)

```

Bien sûr votre code doit fonctionner correctement si on donne d'autres valeurs à x et y aux lignes 1 et 2.

On fera comme lorsqu'on veut échanger deux meubles A et B de place : on utilise un emplacement de stockage temporaire.

Calculs

Faire effectuer par Python les calculs suivants :

$$a = 10^{2 \times 3} \quad b = \frac{666}{74 \times 9} \quad c = 2 \times 3,14159 + 2,71828 + 1,4142 \times 5$$

Priorités des opérateurs

L'expression $2 + 3 * 5$ pourrait a priori s'interpréter $(2 + 3) * 5$ ou $2 + (3 * 5)$. Demandons à Python la valeur des trois expressions :

```
1 x=2 + 3 * 5
2 y=(2 + 3) * 5
3 z=2 + (3 * 5)
4 print (x==y, x==z)
5 False True
```

Donc la bonne interprétation avec les parenthèses est $2 + (3 * 5)$.

Faire de même pour les cinq expressions suivantes autrement dit placer toutes les parenthèses utiles dans l'expression pour obtenir une expression non ambiguë et de même valeur :

```
1 - 1 ** 2
2 10 ** 2 ** 3
3 7 * 29 / 7
4 2 / 1 * 2
5 60 / 6 / 2 / 5
```

La finance aime Python

Traduire en utilisant une variable `milliard` et une variable `million` l'opération suivante :

200 milliards de millions augmenté de 2020 millions et privé de 925 milliards.

Afficher le résultat (on trouvera 199999077020000000).

Vulnerant omnes, ultimat necat

- ① Combien y-a-t-il de secondes dans un siècle ? (utiliser des variables `une_heure`, `j` et `an` et supposer qu'une année est constituée de 365 jours 1/4)
- ② À partir de quel âge (entier) un individu a vécu au moins 1 milliard de secondes (réponse : 32 ans) ?

Diviseur, multiple

Utiliser massivement des variables et les possibilités d'affichage multiple de chaînes et de nombres (cf. cours) pour avoir un code le plus homogène possible.

- ① Vérifier que 101 est un diviseur de 2020.
- ② Montrer que 192642212037519289 et 138633362400520449 ont même reste dans la division entière par 2020.

Multiples de 421

Ecrire la liste des 5 premiers multiples de 421. On donnera deux codes :

- un code affichant les nombres les uns en-dessous des autres,
- un code affichant les nombres les uns à côtés des autres.

Nombre de tours

Utiliser des variables pour résoudre le problème suivant :

un coureur fait un tour de circuit en c minutes. Quel est le nombre x de tours complets qu'il effectuera en n minutes ? Appliquer avec $c = 7$, $n = 240$.

Reproduire un affichage

Soit une variable x , représentant un entier. On cherche à fournir un affichage du calcul de $10x + 1$. Si, par exemple, $x = 42$, on veut que l’affichage ait EXACTEMENT la forme suivante

```
1 x = 42 => 10 * x + 1 = 421
```

ou encore (si $x = 3$) :

```
1 x = 3 => 10 * x + 1 = 31
```

Écrire un code qui produise ce type d’affichage.

Chapitre II

Conditions

1 Booléens

a) Opérateurs logiques ET, OU et NON

Les notions d'*assertion logique* et d'*opérateur logique* existent indépendamment de tout langage de programmation. L'assertion¹ $2k + 1 \geq 100$ est un exemple d'assertion logique (elle est vraie ou fausse selon la valeur de k).

L'opérateur ET

À partir de deux propositions logiques p et q , on peut former la proposition logique p ET q . Par exemple, si on a les deux propositions logiques :

$$p : 5 > 3, q : 4 > 7$$

alors

- p est *vraie*
- q est *fausse*
- p ET q est une proposition *fausse*.

Règle : p ET q est vraie uniquement si p est vraie et si q est également vraie.

Cette règle est conforme à l'usage habituel de la conjonction de coordination ET.

Voici donc la table de vérité de l'opérateur ET :

p	q	p ET q
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Une expression de la forme p ET q s'appelle une *conjonction* et p et q s'appellent les *opérandes* de la conjonction.

1. Au lieu d'*assertion logique*, on rencontre parfois les terme de *proposition* ou de *prédicat*.

La notion d'opérateur OU

À partir de deux propositions logiques p et q , on peut former la proposition logique $p \text{ OU } q$. Par exemple, si

$$p : 5 > 3, q : 4 > 7$$

alors :

- p est vraie
- q est fausse
- $p \text{ OU } q$ est une proposition vraie.

Règle : $p \text{ OU } q$ est fausse uniquement si p est fausse *et* si q est également fausse.

Une expression de la forme $p \text{ OU } q$ s'appelle une *disjonction*. Les assertions p et q s'appellent les *opérandes* de la disjonction.

Voici la table de vérité de l'opérateur OU :

p	q	$p \text{ OU } q$
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Le OU est non exclusif

Le OU logique n'a pas valeur exclusive comme dans *fromage ou dessert* et qui signifie

soit fromage soit dessert mais pas les deux.

Autrement dit, dès que l'une des deux assertions p ou q sont les deux vraies alors l'assertion $p \text{ OU } q$ est vraie.

Par exemple, si

$$p : 5 > 3, q : 4 < 7$$

alors :

- p est vraie,
- q est vraie,
- la proposition $p \text{ OU } q$ est vraie.

L'opérateur logique NON

La proposition NON p est la *négation* de p . Si

$$p : 5 > 3$$

alors :

- p est vraie
- NON p est fausse.

Règle : toute proposition logique p admet une négation q , notée NON p . La valeur logique de q est

- vraie si p est fausse,
- fausse si p est vraie.

b) Les booléens en Python

Les valeurs de vérité *Vrai* et *Faux* en Python sont représentées par deux constantes du langage :

- `True` pour Vrai
- `False` pour Faux

`True` et `False` sont des mots-clés du langage Python 3. Les booléens `True` et `False` appartiennent, comme les entiers ou les flottants, à un type, le type `bool`.

Opérateurs de comparaison et booléen

Certaines expressions ont pour valeur des booléens. C'est le cas, par exemple, de toute expression utilisant les opérateurs de comparaison². Par exemple, l'expression Python `42 > 45` vaut `False` :

```
1 print(42 > 45)
2 print(42 == 40 + 2)
3 False
4 True
```

- Lignes 3-4 : l'assertion « 42 est strictement supérieur à 45 » vaut `False`.
- Ligne 1 : l'assertion « 42 a même valeur que 40 + 2 » vaut `True` (elle est vraie).

Voici un exemple d'utilisation :

```
1 n = 5 * 12 + 6 ** 3 - 42 + 17
2 print(n > 256)
3 print(n)
4 False
5 251
```

- Ligne 1 : `n` est un entier dont la valeur explicite n'apparaît pas dans le code.
- Ligne 2 : On teste pour savoir si l'entier `n` est plus grand que 256.
- Ligne 4 : On lit que non (cf. `False`), `n` n'est pas plus grand que 256.
- Lignes 3 et 5 : le calcul explicite de `n` confirme que l'assertion `n > 256` est fausse.

c) Opérateurs `and`, `or` et `not`

L'opérateur logique ET est représenté en Python par le mot-clé et opérateur `and` :

```
1 p = (5 > 3)
2 q = (4 > 7)
3 print(p)
4 print(q)
5 print(p and q)
6 True
7 False
8 False
```

- Lignes 5 et 8 : `p and q` vaut `False` puisque `q` vaut `False`.

2. Les opérateurs de comparaison numérique sont : `==` `<` `>` `<=` `>=` `!=`

L'opérateur logique OU est représenté en Python par le mot-clé et opérateur `or`.

```
1 p = (5 > 3)
2 q = (4 > 7)
3 print(p)
4 print(q)
5 print(p or q)
6 True
7 False
8 True
```

— Lignes 5 et 8 : `p or q` vaut `True` puisque `p` vaut `True`.

L'opérateur NON est représenté en Python par le mot-clé et opérateur `not`.

```
1 p = (5 == 2 + 3)
2 print(p, not p)
3
4 q = (3 > 4)
5 print(q, not q)
6 True False
7 False True
```

d) Expressions booléennes

De la même façon qu'il existe des expressions arithmétiques comme

$$x*(x+1) - 4*x*y + 1,$$

il existe des *expressions booléennes*, comme

$$p \text{ and } (q \text{ or } r) \text{ or } (\text{not } p \text{ and } q),$$

qui s'obtiennent en combinant des opérateurs logiques avec des expressions.

Évaluer une *expression booléenne*, c'est déterminer si elle vaut `True` ou `False`.

Par exemple

$$(1 == 2 \text{ and } 2 == 3) \text{ or } 1 == 1$$

est une expression booléenne dont la valeur est `True`.

Bien se rendre compte que dans de nombreuses situations, les parenthèses figurant dans l'expression sont indispensables. Ainsi, comparer :

```
1 a=1
2 b=2
3 print((not a) == b)
4 print(not a == b)
5 False
6 True
```

Pour conclure, donnons deux exemples typiques d'expressions booléennes.

Exemple 1 : être positif

Il est fréquent que les expressions booléennes utilisent des variables. Par exemple, si `x` est une variable, on peut définir une expression booléenne qui exprime si oui ou non `x` représente un nombre positif : `x >= 0`. On peut référencer ces expressions booléennes elles-mêmes par une variable. Ainsi :

```
1 x = 2**19 - 10**6
2 estPos = (x >= 0)
3
4 print(estPos)
5 print(x)
6
7 vrai = True
8 print(estPos == vrai)
9 False
10 -475712
11 False
```

- Ligne 2 : on a défini une « variable booléenne » `estPos` et qui ici permet de tester le caractère positif de n'importe quel nombre `x` défini au préalable. L'usage des parenthèses n'est pas obligatoire, autrement dit, l'écriture `estPos = x >= 0`, bien que moins lisible, est licite.
- Ligne 4 : l'affichage (ligne 10) montre que `x` est négatif donc la valeur de la variable `estPos` est bien `False` (ligne 9).
- Ligne 6 : on peut placer la constante `True` dans une variable. Idem pour la constante `False`.
- Ligne 7 : si `b` est une variable booléenne, l'expression `b == True` a toujours même valeur que `b`, autrement dit, en pratique, il n'est que rarement justifié d'écrire une expression de la forme `b == True`.

Exemple 2 : comparer des heures

On donne deux moments de la journée, désignés par `M` et `MM`, en heures et minutes, par exemple

`M = 14 h 52 min` et `MM = 14 h 51 min`.

et on demande de dire si, oui ou non, `M` est un moment de la journée strictement antérieur à `MM`. Dans l'exemple ci-dessus, la réponse est *non*.

Chaque moment sera donné sous la forme de deux entiers `h`, `m` représentant les heures et les minutes pour `M` et de même `hh`, `mm` pour `MM`.

Pour exprimer que `M` est antérieur à `MM`, on compare les heures : `h < hh`. Si elles sont distinctes, on peut conclure, sinon c'est que l'expression booléenne `h == hh` est `True`, et il faut comparer les minutes (`m < mm`) pour conclure. D'où l'expression booléenne suivante :

`(h < hh) or (h == hh and m < mm)`

Cette expression est vraie si et seulement si `M` est strictement antérieur à `MM`. Voici un exemple d'utilisation :

```

1 h=14
2 m=52
3
4 hh=14
5 mm=51
6
7 print((h < hh) or (h == hh and m < mm))
8 False

```

EXERCICES TYPE

Egaux ou opposés

Ecrire une variable booléenne `egaux_ou_opp` qui à partir de deux entiers `x` et `y` teste si les deux nombres sont égaux ou opposés. Exemples de comportement :

x	y	egaux_ou_opp
5	6	False
42	42	True
42	-42	True

Solution

Deux entiers `x` et `y` sont égaux s'il vérifient la condition

$$A : x == y$$

et ils sont opposés s'ils vérifient la condition

$$B : x == -y.$$

Donc `x` et `y` sont égaux ou opposés si la condition `A or B` est vraie. Cette condition peut être placée dans une variable booléenne `egaux_ou_opp`. D'où le code suivant :

```

1 x = 42
2 y = -42
3 egaux_ou_opp = (x == y or x == -y)
4 print(x, y, "->", egaux_ou_opp)
5 print('-----')
6
7 x = 42
8 y = 42
9 egaux_ou_opp = (x == y or x == -y)
10 print(x, y, "->", egaux_ou_opp)
11 print('-----')
12
13 x = 421
14 y = 42
15 egaux_ou_opp = (x == y or x == -y)
16 print(x, y, "->", egaux_ou_opp)

```

```

17 42 -42 -> True
18 -----
19 42 42 -> True
20 -----
21 421 42 -> False

```

- Trois types d'exemples sont testés (lignes 1-2, lignes 7-8 et lignes 13-14) qui répètent toujours le même code.
- Lignes 3, 9 et 15 : la variable `egaux_ou_opp` vaut `True` si `x` et `y` sont égaux ou opposés et dans tous les autres cas, elle vaut `False`.

Complément

On pouvait aussi remarquer que des nombres égaux ou opposés, par exemple 10 et 10 ou encore 10 et -10 se reconnaissent parce qu'ils ont même carré, dans notre exemple ce serait 100. D'où le code suivant, en apparence plus simple :

```

1 x = 42
2 y = -42
3 egal_ou_opp = (x**2 == y**2)
4 print(x, y, "->", egal_ou_opp)
5 print('-----')
6
7 x = 42
8 y = 42
9 egal_ou_opp = (x**2 == y**2)
10 print(x, y, "->", egal_ou_opp)
11 print('-----')
12
13 x = 421
14 y = 42
15 egal_ou_opp = (x**2 == y**2)
16 print(x, y, "->", egal_ou_opp)

```

```

17 42 -42 -> True
18 -----
19 42 42 -> True
20 -----
21 421 42 -> False

```

Ce code a toutefois deux inconvénients :

- alors que le code Python précédent serait transposable dans d'autres langages comme Java, C++ ou Javascript, le présent code ne le serait plus car le code contient un risque d'*overflow*. En effet, si x est un entier alors le carré de x est beaucoup plus grand et sa taille pourrait dépasser la taille-machine admise pour un entier (cette situation n'est pas possible en Python qui gère nativement les grands entiers).
- le résultat se fait au prix du calcul de deux carrés ce qui a un certain coût. Si ces calculs devaient se répéter (dans des boucles par exemple), cela pourrait avoir un impact sur les performances du programme.

Une solution moins coûteuse tout en étant transposable à d'autres langages serait d'utiliser la fonction valeur absolue : en effet, deux nombres sont égaux ou opposés si et seulement s'ils ont

même valeur absolue. En Python, la fonction valeur absolue est accessible nativement (fonction `abs`), d'où le code :

```
1 x = 42
2 y = -42
3 egaux_ou_opp = (abs(x) == abs(y))
4 print(x, y, "->", egaux_ou_opp)
```

Année bissextile

Une année bissextile est une année multiple de 4, non multiple de 100 à moins qu'elle ne soit multiple de 400. Ainsi,

- 2017 n'est pas bissextile puisque 2017 n'est pas un multiple de 4 ;
- 2020 est bissextile car elle est multiple de 4 et non multiple de 100 ;
- 3000 n'est pas bissextile car elle est multiple de 100 sans être multiple de 400 ;
- 2000 était bissextile car 2000 est multiple de 400.

Considérons les trois booléens suivants :

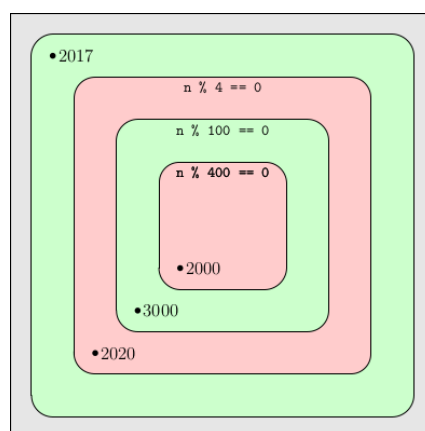
```
1 p = (n % 4) == 0
2 q = (n % 100) == 0
3 r = (n % 400) == 0
```

Exprimez à l'aide d'une expression booléenne dépendant de `p`, `q` et `r` que l'année `n` est bissextile.

Solution

Le dessin ci-dessous illustre la répartition des années bissextiles (en rouge). Une année `n` est bissextile exactement si

- elle est multiple de 4 ET
- elle n'est pas un multiple de 100 OU elle est multiple de 400



Notons les traductions suivantes :

- variable `p` : `n` est multiple de 4,

- variable `q` : `n` est multiple de 100,
- variable `r` : `n` est multiple de 400.

Donc, avec les variables de l'énoncé, une année `n` est bissextile si et seulement si le booléen³

`estBissextile = p and (not q or r)`

vaut `True`.

D'où le code suivant qui traite les quatre cas possibles :

```

1 n=2017
2 p = (n % 4)==0
3 q = (n % 100)==0
4 r = (n % 400)==0
5 estBissextile = p and (not q or r)
6 print(n, "->", estBissextile)
7 print("-----")
8
9 n=2020
10 p = (n % 4)==0
11 q = (n % 100)==0
12 r = (n % 400)==0
13 estBissextile = p and (not q or r)
14 print(n, "->", estBissextile)
15 print("-----")
16
17 n=3000
18 p = (n % 4)==0
19 q = (n % 100)==0
20 r = (n % 400)==0
21 estBissextile = p and (not q or r)
22 print(n, "->", estBissextile)
23 print("-----")
24
25 n=2000
26 p = (n % 4)==0
27 q = (n % 100)==0
28 r = (n % 400)==0
29 estBissextile = p and (not q or r)
30 print(n, "->", estBissextile)
31 print("-----")
32 2017 -> False
33 -----
34 2020 -> True
35 -----
36 3000 -> False
37 -----
38 2000 -> True
39 -----

```

3. Cette façon de coder le caractère bissextile est largement « idiomatique » ; c'est ainsi que la bibliothèque standard le code : [2_isleaps](#).

Au passage, on pourra remarquer **ici** qu'il est équivalent d'écrire :

```
estBissextile = (p and not q) or r
```

(ce genre d'équivalence n'est pas toujours vrai mais elle l'est ici car un multiple de 400 est forcément un multiple de 4).

Alternative

Il aurait aussi été possible d'écrire un booléen `estNonBissextile` traduisant qu'une année `n` est non bissextile en disant qu'elle n'est pas multiple de 4 (traduction : `not p`) ou alors (traduction `or`) qu'elle est multiple de 100 et pas multiple de 400 (traduction : `q and not r`) puis en prenant la négation de `estNonBissextile` :

```
1 n=3000
2 p = (n % 4)==0
3 q = (n % 100)==0
4 r = (n % 400)==0
5 estNonBissextile =(not p) or (q and not r)
6 estBissextile =not estNonBissextile
7
8 print(n, "->", estBissextile)
```

Mois de 31 jours

On donne un numéro de mois entre 1 et 12. Créer une variable booléenne `est_mois_31` (prenant comme valeur `True` ou `False`) qui teste si `m` est le numéro d'un mois ayant 31 jours ^a comme janvier (numéro 1) ou juillet (numéro 7) mais pas février (numéro 2).

^a. Les mois ayant 31 jours sont : janvier (1), mars (3), mai (5), juillet (7), août (8), octobre (10), décembre (12).

Solution

On dispose de la liste des numéros de mois à 31 jours, donc il suffit de tester les uns après les autres le mois donné `m` avec les mois possibles :

```
1 m = 4
2 est_mois_31 = (m == 1 or m == 3 or m == 5 or
3               m == 7 or m == 8 or m == 10 or m == 12)
4
5 print(est_mois_31)
6 False
```

— Lignes 2-3 : une expression placée entre des parenthèses peut être exceptionnellement écrite sur plusieurs lignes, cela évite d'écrire de longues lignes.

On peut aussi essayer de donner une condition plus synthétique. Un mois de 31 jours placé avant août est de numéro 1, 3, 5 ou 7 et donc impair ; à l'inverse, un mois de 31 jours placé à partir d'août est de numéro 8, 10 ou 12 et donc pair. D'où le code suivant :

```

1 m = 9
2 est_mois_31 = ((m <= 7 and m % 2 == 1) or (m >= 8 and m % 2 == 0))
3 print(est_mois_31)

```

4 False

On pouvait aussi remarquer qu'il y a plus de mois à 31 jours et donc il est (légèrement) plus court d'écrire de passer par la négation :

```

1 m = 4
2 est_mois_31 = not (m==2 or m==4 or m==6 or m==9 or m==11)
3 print(est_mois_31)

```

4 False

14 juillet

Une date est codée avec deux variables *j* et *m* où *j* représente le jour du mois et *m* le numéro du mois. Par exemple, le 2 novembre est codé avec *j* = 2 et *m* = 11. On vous donne deux variables *j* et *m* représentant une date valide et vous devez construire une variable booléenne notée *avant* et qui vaut **True** si la date donnée tombe le 14 juillet ou les jours d'avant et **False** si elle tombe à partir du 15 juillet. Ci-dessous, quelques exemples d'exécution du programme demandé :

```

1 (10, 4) -> True
2 (15, 6) -> True
3 (10, 7) -> True
4 (14, 7) -> True
5 (20, 7) -> False
6 (2, 11) -> False
7 (25, 12) -> False

```

Solution

Pour savoir si une date (*j*, *m*) est antérieure au 14 juillet, on commence par regarder si le mois *m* vérifie *m* < 7 et si c'est le cas, la date est bien antérieure ; sinon, on regarde si *m* vaut 7 et dans ce cas, la date sera antérieure selon que le jour sera ou pas, antérieur au 14. Dans tous les autres cas, la date n'est pas antérieure. D'où le code :

```

1 j = 10
2 m =4
3 avant = m<7 or m==7 and j<=14
4 print((j,m), "->", avant)
5
6 j = 15
7 m =6
8 avant = m<7 or m==7 and j<=14
9 print((j,m), "->", avant)
10
11 j = 10
12 m =7
13 avant = m<7 or m==7 and j<=14
14 print((j,m), "->", avant)
15
16 j = 14
17 m =7
18 avant = m<7 or m==7 and j<=14
19 print((j,m), "->", avant)
20
21 j = 20
22 m =7
23 avant = m<7 or m==7 and j<=14
24 print((j,m), "->", avant)
25
26 j = 2
27 m =11
28 avant = m<7 or m==7 and j<=14
29 print((j,m), "->", avant)
30
31 j = 25
32 m =12
33 avant = m<7 or m==7 and j<=14
34 print((j,m), "->", avant)
35 (10, 4) -> True
36 (15, 6) -> True
37 (10, 7) -> True
38 (14, 7) -> True
39 (20, 7) -> False
40 (2, 11) -> False
41 (25, 12) -> False

```

EXERCICES

Conditions sur un entier

Dans cette question, x est supposé être un entier. Créer une variable booléenne `cond` (prenant comme valeur `True` ou `False`) qui teste si x est ou bien multiple de 10 ou bien à la fois impair, non multiple de 3 et compris, au sens large, entre 42 et 421. Ainsi $x = 2020$ passe le test, $x = 301$ aussi, $x = 420$ également mais pas $x = 81$.

Vacances

On donne deux entiers j et m représentant une date valide où m représente le numéro du mois (entre 1 et 12) et j le jour du mois (entre 1 et 31). Ainsi,

- la donnée $j = 25$ et $m = 4$ représente le 25 avril
- le 31 septembre est représenté par $j = 31$ et $m = 9$.

En France, les vacances d'été 2017 auront lieu du dimanche 8 juillet au dimanche 3 septembre 2017. On donne le jour j et le mois m d'une date et on vous demande de construire une variable booléenne `enVacances` valant `True` si la date correspond à un jour de vacances d'été 2017 et `False` sinon. Exemples :

```
1 j = 1 et m = 5 -> False
2 j = 6 et m = 7 -> False
3 j = 7 et m = 7 -> False
4 j = 14 et m = 7 -> True
5 j = 2 et m = 8 -> True
6 j = 31 et m = 8 -> True
7 j = 3 et m = 9 -> True
8 j = 4 et m = 9 -> False
9 j = 7 et m = 9 -> False
10 j = 8 et m = 12 -> False
```

Test de la non-égalité de cinq entiers

On donne cinq entiers a , b , c , d et e . Soit la condition C suivante : *les 5 entiers ne sont pas tous égaux*. Par exemple

- si $a=5$, $b=5$, $c=5$, $d=4$ et $e=4$ alors C est `True`
- si $a=5$, $b=5$, $c=5$, $d=5$ et $e=5$ alors C est `False`

Ecrire une variable booléenne qui vaut `True` si la condition C est vraie et `False` sinon.

Au moins un pair et au moins un supérieur à 100

On donne trois entiers positifs a , b et c . Construire un booléen `ok` qui vaut `True` si parmi les trois nombres, il y a au moins un nombre pair et aussi s'il y a au moins un nombre supérieur ou égal à 100. Sinon, `ok` vaut `False`

Exemples :

```
1 7, 42, 142 -> True
2 42, 60, 71 -> False
3 111, 125, 2015 -> False
4 80, 111, 31 -> True
```

Entiers qui se suivent

On vous donne deux variables x et y représentant des entiers. Définir une variable booléenne `ok` valant `True` si

- les entiers sont entre 1 et 4 ET
- ou bien sont consécutifs
- ou bien l'un vaut 1 et l'autre vaut 4.

Sinon, `ok` vaudra `False`.

Exemples

```
1 5 2 -> False
2 2 3 -> True
3 3 2 -> True
4 4 4 -> False
5 7 8 -> False
6 4 1 -> True
7 3 1 -> False
```

Formule

On connaît la formule mathématique suivante $1 + x + x^2 + x^3 = \frac{x^4 - 1}{x - 1}$.

- 1 Introduire une variable `x`, par exemple `x = 42` et traduire les deux expressions (celle de gauche et celle de droite) de la formule ci-dessus par du code Python avec deux variables `gauche` et `droite`. ATTENTION aux parenthèses!
Afficher un booléen qui indique si les valeurs de `gauche` et `droite` sont égales.
- 2 Vérifiez l'exactitude de cette formule pour `x = 208066`. Si la formule n'est pas exacte, modifiez votre code en observant que le quotient exact au membre de droite doit être un entier puisque le membre de gauche est un entier.

Entier ayant n chiffres

On vous donne un entier $x > 0$ et un autre entier $n > 0$. Créer une variable booléenne `avoir_n_chiffres` (prenant comme valeur `True` ou `False`) qui teste si l'entier x admet exactement n chiffres dans son écriture en base 10. Par exemple, si $x = 421$ et $n = 2$, la condition vaut `False` et si $n = 3$, la condition vaut `True`. Pour trouver la condition, on pourra observer, en utilisant des puissances de 10, quel est le plus petit et le plus grand entier ayant n chiffres.

2 Instructions conditionnelles

a) Instruction `if`

L'instruction `if` permet de coder un schéma logique du type

si ... alors ...

Illustration

On veut coder un programme qui :

- définit une variable `x` référant un entier,
- affiche la valeur de `x`,
- teste si `x` est strictement négatif et, si c'est le cas, affiche :
 - le message : **`x` est négatif**,
 - l'opposé de `x` (c'est-à-dire `-x`),
- se termine en affichant le message **Fin du programme**.

Voilà ce que pourrait afficher le programme si la variable `x` dans le code-source vaut 10 :

```
1 -10
2 x est négatif
3 10
4 Fin du programme
```

Le programme sera intitulé `instruction_if.py`.

Implémentation en Python

Le programme `instruction_if.py` doit afficher un message et un nombre **si** une certaine condition est vérifiée. Pour cela, en Python, on utilise

- une instruction **if** pour exprimer l'idée de condition *si ... alors ...*
- un booléen pour exprimer la condition *si*.

Le problème posé admet la solution suivante :

`instruction_if1.py`

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

```
9 -10
10 x est négatif
11 10
12 Fin du programme
```

Examinons maintenant le même code que `instruction_if1.py` où on a remplacé `x = -10` par `x = 42` en ligne 1 :

`instruction_if2.py`

```
1 x = 42
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

```
9 42
10 Fin du programme
```

L'édition d'une instruction `if`

On reprend le code ci-dessus :

instruction_if1.py

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

- Lignes 4-6 : une instruction conditionnelle. Elle utilise le mot-clé **if**.
- Ligne 4 : la condition testée est $x < 0$. Cette condition est toujours suivie du séparateur : (*deux-points*). La ligne qui s'étend du mot-clé **if** jusqu'au séparateur s'appelle l'*en-tête* de l'instruction **if**.
- Lignes 5-6 : le *corps* de l'instruction **if** est le bloc situé aux lignes 5 et 6. Ce corps, la plupart du temps :
 - commence **sous** l'en-tête (il faut donc sauter une ligne);
 - est indenté par rapport à l'en-tête.

Exécution d'une instruction if

On reprend les codes ci-dessus :

instruction_if1.py

```
1 x = -10
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

ainsi que

instruction_if2.py

```
1 x = 42
2 print(x)
3
4 if x < 0:
5     print("x est négatif")
6     print(-x)
7
8 print("Fin du programme")
```

- Ligne 1 : très souvent, la condition de l'instruction **if** dépend d'une variable définie **avant** l'instruction **if**. Ici, la variable s'appelle **x**.
- Ligne 4 : lorsque l'exécution du code arrive face à une condition **if**, la condition **if** est testée : vaut-elle **True** ? vaut-elle **False** ?
- Lignes 5-6 :
 - si la condition vaut **True** (cf. `instruction_if1.py`), les instructions du **corps** de **if** sont exécutées

- si la condition vaut `False` (cf. `instruction_if2.py`), les instructions du corps de `if` sont ignorées et l'exécution passe directement à la suite du code (ligne 8)
- Ligne 8 : Une fois l'instruction `if` exécutée, l'exécution continue normalement.

b) Indentation d'une instruction composée

Soit le code d'une instruction `if` typique :

```
1 x = -100
2
3 if x < 0:
4     print("x est negatif")
5     print(-x)
6
7 print("Fin du programme")
```

Le programme ci-dessus contient trois instructions :

- Ligne 1 : une instruction simple (affectation)
- Lignes 3-5 : l'instruction `if` qui est une instruction composée.
- Ligne 7 : une instruction simple (instruction d'affichage)

Ces trois instructions sont indentées avec la même valeur d'indentation qui est de 0 espace par rapport à la marge gauche du code.

On observe toutefois (lignes 4-5) que certaines lignes du code sont indentées.

Règles fondamentales de l'indentation

L'instruction `if` (lignes 3-5) est une instruction *composée* et son indentation doit respecter deux règles fondamentales :

- Règle 1 : si le **corps** de l'instruction `if` est placé sur plusieurs lignes, ce corps est indenté par rapport à la ligne d'en-tête ;
- Règle 2 : toutes les instructions dans le corps de l'instruction sont à la même indentation. Autrement dit, si par exemple la 1^{er} instruction du corps de l'instruction `if` (ici ligne 4) est indentée de 4 espaces par rapport à l'en-tête, toutes les autres instructions du corps de l'instruction `if`, par exemple ici la ligne 5, seront aussi indentées de 4 espaces par rapport à l'en-tête.

Non-respect des règles

Règle 1

Le code ci-dessous ne respecte pas la règle 1 :

```
1 x = -100
2
3 if x < 0:
4     print("x est negatif")
5     print(-x)
6
7 print("Fin du programme")
```

— Lignes 4-5 : ces deux lignes devraient être indentées.

et l'exécution va générer un message d'erreur d'indentation au niveau de la ligne 4 :

```
1 File "if_erreur1_indentation.py", line 4
2     print("x est negatif")
3     ^
4 IndentationError: expected an indented block
```

— Lignes 8-11 : Le message d'erreur.

— Lignes 8 : Le nom du fichier-source⁴ et le numéro de ligne à l'origine de l'erreur sont fournis.

— Lignes 9-10 : La ligne à l'origine de l'erreur et un marqueur (le signe circonflexe ^) signalant où l'erreur est rencontré dans la ligne.

— Ligne 11 : Le type d'erreur, ici **IndentationError** (*erreur d'indentation*) et une description sommaire de la raison de cette erreur.

Règle 2

Le code ci-dessous ne respecte pas la règle 2 :

if_erreur2_indentation.py

```
1 x = -100
2
3 if x < 0:
4     print("x est negatif")
5     print(-x)
6
7 print("Fin du programme")
```

— Lignes 4-5 : ces deux lignes devraient être à un même niveau d'indentation. Or la ligne 4 est indentée de 5 espaces et la ligne suivante de 4 espaces au lieu de 5.

et l'exécution va générer un message d'erreur d'indentation au niveau de la ligne 5 :

if_erreur2_indentation.py

```
1 File "if_erreur2_indentation.py", line 5
2     print(-x)
3     ^
4 IndentationError: unindent does not match any outer indentation level
```

— Ligne 11 : Le message explique que l'indentation n'est pas uniforme. L'indentation de la ligne 4 est acceptée pas celle de la ligne 5.

c) Instructions **if** imbriquées

Le corps d'une instruction **if** peut être n'importe quelle instruction, en particulier une nouvelle instruction **if**.

Exemple

4. En fait, cela dépend de l'interface d'utilisation de Python. Pour une feuille IPython/Jupyter, aucun nom de fichier n'est mentionné.

```

1 n = 3000
2 if n % 10 == 0:
3     q = n // 10
4     if q % 10 == 0:
5         print("n est multiple de 100")
6 n est multiple de 100

```

Une instruction `if` est une instruction multiple. Le corps d'une instruction `if` peut être de taille quelconque et contenir tout type d'instructions.

d) Instruction if/else

L'instruction if/else permet de coder un schéma du type

si ... alors ... sinon ...

Soit un programme qui à partir d'un nombre x affiche *plus* si $x \geq 42$ et qui sinon affiche *moins*. En Python, cela donne :

```

if_else.py
1 x=10
2 if x > 42:
3     print("plus")
4 else:
5     print("moins")
6
7 print("FIN")
8 moins
9 FIN

```

- Ligne 2 : comme dans une instruction `if` simple, la condition est testée
- Lignes 2-5 : l'instruction `if/else`
- Ligne 4 : si la condition N'est PAS vérifiée,
 - la partie sous le `if` N'est PAS examinée
 - le bloc sous le `else` est exécuté
- Ligne 7 : l'exécution du programme continue alors normalement.

Inversement, si la condition suivant le `if` est vérifiée la partie `else` n'est pas examinée :

```

if_else_bis.py
1 x=50
2 if x > 42:
3     print( "plus")
4 else:
5     print("moins")
6
7 print("FIN")
8 plus
9 FIN

```

- Ligne 2 : ici, la condition $x > 42$ est vérifiée donc seule la ligne 3 est examinée

— Lignes 4-5 : ces lignes ne sont pas examinées durant l'exécution.

Le mot **else** doit être compris comme *sinon*. Une instruction **if/else** propose une **alternative**. Ainsi, le programme `if_else.py` doit être compris comme codant l'alternative suivante :

— si `x > 42` alors le programme affiche **plus**

— sinon, le programme affiche **moins**

Le mot **else** est un mot-clé du langage Python. Dans le cadre d'une instruction **if/else**, le mot **else** doit toujours être suivi du signe deux-points (`:`). Sous la ligne figurant **else** se trouve un bloc indenté. La ligne contenant **else** ainsi que le bloc indenté forment ce qu'on appelle une *clause else* de l'instruction **if**. Le *corps* de la clause **else** est le bloc indenté. Ce corps peut être constitué de n'importe quel type d'instructions.

C'est une erreur courante chez les débutants de vouloir placer une condition après le `else`, par exemple `else x<42`. Une clause **else** est toujours suivie immédiatement du signe `:`.

Quand on écrit une instruction `if/else`, prendre garde que le mot `if` et le mot `else` soient à la même indentation.

Le terme d'instruction **if/else** est un abus de langage. En fait, il s'agit d'une instruction **if** contenant une clause **else**.

e) Indentation d'une instruction composée avec clause

Une instruction composée est constituée au minimum d'un en-tête et d'un corps, typiquement :

```
1 if 42 > 421:
2     print("hello!")
3     print("OK!")
```

— Ligne 1 : l'en-tête

— Lignes 2-3 : corps composé de deux instructions.

Mais, certaines instructions composées peuvent être composées aussi d'une, voire de plusieurs, « clauses ».

Le cas suivant d'une instruction `if/else` est typique de la structure générale d'une instruction composée avec clause :

```
1 x=50
2 if x > 42:
3     print( "plus")
4     if x > 100:
5         print("plus plus")
6 else:
7     print(x)
8     print("moins")
9
10 print("FIN")
```

— Lignes 2-8 : une instruction **if**

— Lignes 6-8 : une clause `else` de l'instruction **if**

— Lignes 3-5 : le corps de l'instruction `if`. Ce corps est constitué de deux instructions : une instruction simple (ligne 3) et une instruction composée (`if`, lignes 4-5)

— Lignes 6-8 : le corps de la clause else.

Les clauses font partie de l'instruction composée. Chaque clause contient un en-tête placée sur une seule ligne et qui se termine toujours par le signe deux-points. Le signe deux-points est suivi d'un ensemble d'instructions appelé « corps » de la clause.

Règles d'indentation

Soit une instruction composée contenant une ou plusieurs clauses. Alors,

- l'en-tête de l'instruction composée et chacune des clauses sont sur des lignes différentes et à la même indentation ;
- les instructions figurant dans le corps de l'instruction composée sont à la même indentation ;
- les instructions figurant dans le corps de n'importe quelle clause de l'instruction composée sont à la même indentation ;
- le corps de l'instruction composée ou de l'une de ses clauses peut contenir des instructions de n'importe quel type, simples ou composées. Ces instructions suivent les règles générales de l'indentation en Python.

f) Indentation intelligente des éditeurs

L'indentation d'une ligne code est marquée par un décalage vertical par rapport à la marge.

Dans la plupart des éditeurs, l'indentation est intelligente et constante :

- intelligente car l'éditeur, en fonction de la présence du séparateur deux-points en fin de ligne, sait s'il doit, ou non, indenter la ligne suivante
- constante car chaque passage à la ligne dans un bloc fait commencer la ligne à la même indentation que la précédente.

Soit le code suivant :

```
1 x = -100
2
3 if x < 0:
4     print("x est negatif")
5     print(-x)
6
7 print "Fin du programme"
```

- Lignes 1 et 3 : pas d'indentation.
- Ligne 4 : le début de ligne est indenté automatiquement.
- Ligne 5 : le début de ligne est positionné automatiquement.
- Ligne 6 : l'éditeur indente automatiquement le début de ligne et le codeur doit retirer lui-même l'indentation avec la touche **Retour-Arrière**.

Dans un éditeur adapté au langage Python, le programmeur n'a pas à gérer l'indentation de son code. En fait, ce que l'éditeur ne gère pas automatiquement, c'est *la fin de l'indentation*, autrement dit le moment où le bloc indenté se termine et qu'il faut diminuer l'indentation, ce qui dépend seulement du souhait du programmeur et que l'éditeur ne peut deviner.

g) Instruction `if/elif`

L'instruction `if/elif` permet de coder un schéma du type

si ... alors ... sinon si ...

Voici un exemple typique d'utilisation d'une instruction `if/elif`

```
if_elif.py
1 m = 2
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")
12
13 2
14 Argent
15 bonjour !
```

L'exécution est la suivante :

- Au départ, `m` vaut 2 et la valeur de `m` est affichée
- Le programme passe à la ligne 4 et teste (à cause du `if`) si `m` vaut 1.
- Comme ce n'est pas le cas, le programme passe à la ligne 6 (à cause du `elif`) et teste si `m` vaut 2.
- Comme c'est le cas, la ligne 7 est exécutée et **Argent** est affiché.
- Les lignes 8-9 sont alors ignorées.
- Finalement, l'exécution passe à la ligne 11.

Le mot `elif` est un mot-clé du langage Python.

Une clause `elif` (lignes 6-7 ou encore 8-9) est constituée de deux parties :

- un en-tête placé sur une ligne et terminé par deux-points
- un corps, le plus souvent indenté par rapport à l'en-tête.

Le mot-clé `elif` d'une instruction `if` est toujours aligné verticalement avec le mot-clé `if`.

Les clauses `elif` constituent des blocs de code qui sont à la même indentation que le corps de l'instruction `if`.

`elif` est un mot-valise construit de la manière suivante :

`elif` = `else` + `if`.

Il faut comprendre `elif` comme voulant signifier *sinon si*.

On dit que les parties du code contenant un bloc `elif` sont des *clauses elif*. Parler d'instruction `if/elif` est un abus de langage, il s'agit en fait d'une instruction `if` contenant une ou plusieurs clauses `elif`.

h) Suite de `if` vs suite de `if/elif`

Comparons une suite d'instructions `if/elif` à une suite d'instructions où `elif` est remplacé par `if`.

Par exemple, comparons les codes suivants :

`if_elif.py`

```
1 m = 1
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")
```

`suite_if.py`

```
12 m = 1
13 print(m)
14
15 if m==1:
16     print("Or")
17 if m==2:
18     print("Argent")
19 if m==3:
20     print("Bronze")
21
22 print("bonjour!")
```

Les deux codes affichent la même chose :

```
1 1
2 Or
3 bonjour !
```

Pourtant, les deux exécutions de code ne sont pas équivalentes :

- code `if_elif.py` : les conditions lignes 6 et 8 ne sont même pas examinées car les lignes 4-5 sont exécutées ;
- code `suite_if.py`, les trois conditions (lignes 15, 17 et 19) sont **toujours** testées ce qui n'est pas pertinent puisque les différentes conditions testées s'excluent mutuellement.

Il se pourrait d'ailleurs que dans certains cas, les affichages ne soient pas identiques. Pour cela comparer les deux codes suivants.

Premier code

```
1 m=0
2 if m>=0:
3     print("m est positif")
4 if m<=0:
5     print("m est negatif")
```

```

6 m est positif
7 m est negatif

```

- Lignes 2-3 : l'affichage est exécuté car $0 \geq 0$ est **True**
- Lignes 4-5 : l'affichage est exécuté car $0 \leq 0$ est aussi **True**
- Lignes 6-7 : deux lignes sont affichées

Deuxième code :

```

1 m=0
2 if m>=0:
3     print("m est positif")
4 elif m<=0:
5     print("m est negatif")
6 m est positif

```

- Lignes 2-3 : l'affichage est exécuté car $0 \geq 0$ est **True**
- Ligne 4 : la condition n'est pas testée car la condition ligne 2 a été testée
- Ligne 6 : une seule ligne est affichée.

i) Les conditions **if/elif**

Exclusion mutuelle ?

Soit le code `if_elif.py` :

if_elif.py

```

1 m = 2
2 print(m)
3
4 if m == 1:
5     print("Or")
6 elif m == 2:
7     print("Argent")
8 elif m == 3:
9     print("Bronze")
10
11 print("bonjour !")

```

Les différentes conditions **if/elif** du code `if_elif.py` (ici $m == 1$, $m == 2$, $m == 3$) s'excluent mutuellement ce qui est le cas le plus fréquent en pratique ; mais ce n'est pas obligatoire. Par exemple, soit l'extrait de code suivant qui teste le signe d'un nombre x :

```

1 if x>=0:
2     print("positif")
3 elif x<= 0:
4     print("negatif")

```

Ici, les conditions $x \geq 0$ et $x \leq 0$ ne sont pas incompatibles puisque $x=0$ vérifie les deux. Si on exécute le code avec $x=0$, on obtient :

```

1 x=0
2 if x>=0:
3     print("positif")
4 elif x<= 0:
5     print("negatif")
6 positif

```

Comme la condition ligne 2 est vérifiée, la ligne 4 n'est pas testée.

Cas où aucun corps `if/elif` n'est exécuté

Il se peut d'autre part que le corps d'AUCUNE instruction `if` ou `elif` ne soit exécuté, par exemple :

```

1 m=5
2 if m==1:
3     print ("Or")
4 elif m==2:
5     print("Argent")
6 elif m==3:
7     print("Bronze")
8
9 print("bonjour !")
10 bonjour !

```

- La ligne 2 est testée.
- Comme la condition est fausse, la ligne 3 est sautée
- L'instruction `elif` ligne 4 est testée. A nouveau, elle est fausse
- Donc, l'instruction `elif` de la ligne 6 est testée. Comme elle est encore fausse, la ligne 7 est sautée et aucun nom de médaille n'est affiché.
- L'exécution passe à la ligne 9.

j) Instruction `if/elif/else`

L'instruction `if/elif/else` correspond au schéma suivant :

si ... alors ... sinon si ... alors sinon

Il est possible d'associer à une instruction `if`

- un ou plusieurs `elif`,
- un `else` unique.

Voici un exemple basique

```

1 m=5
2 if m==1:
3     print("Or")
4 elif m==2:
5     print("Argent")
6 elif m==3:
7     print("Bronze")
8 else :
9     print("Sans médaille")
10
11 print("FIN")

```

```

12 Sans médaille
13 FIN

```

- Ligne 1 : l'exécution de l'instruction **if/elif/else** aux lignes 2-9 dépend de la valeur initiale de `m`.
- Lignes 2-7 : les conditions ligne 2, ligne 4 et ligne 6 sont testées dans cet ordre tant qu'elles sont fausses. Le corps de la première (éventuelle) condition vraie est exécuté.
- Lignes 8-9 : si aucune des conditions n'est vraie alors le corps du **else** (ligne 9) est exécuté. C'est le cas ici puisque `m = 5` (cf. la ligne 1).

La clause **else** est unique et doit apparaître **après** les clauses **elif**.

A la différence d'une instruction **if** simple ou d'une instruction **if/elif**, dans une instruction **if/elif/else**, comme dans une instruction **if/else**, au moins un des corps de l'instruction sera exécutée.

k) Parenthèse autour du booléen suivant un **if**

Bien que non fautif, il n'est pas utile de placer des parenthèses autour du booléen testé dans une instruction **if** :

```

1 x = 42
2
3 if (x < 0):
4     print(-x)

```

- Les parenthèse autour de `x < 0` sont inutiles.

le code précédent est équivalent à

```

1 x = 42
2
3 if x < 0:
4     print(-x)

```

Cette pratique semble venir de langages comme C ou Java.

SCREENCAST

EXERCICES TYPE

Régler un montant avec des coupures de même valeur

On vous donne la valeur b d'un billet en euros, par exemple $b = 10$ et un montant m , nombre entier représentant un montant en euros à régler, par exemple, $m = 8181$. Déterminer le nombre minimum N de billets de b euros pour être en mesure de régler le montant m .

Par exemple,

- si $b = 10$ et $m = 8181$ alors $N = 819$
- si $b = 10$ et $m = 800$ alors $N = 80$

Solution

Il y a deux cas selon que le montant peut être réglé de manière exacte avec des billets de montant b . Par exemple, si $b=10$ et $m=8181$, le montant ne peut être réglé de manière exacte (sans rendu de monnaie). Le nombre exact de billets est clairement 819 car avec 818 billets on peut régler jusqu'à 8180 euros (et il manque 1 euro) et avec un billet de plus, on peut régler le montant (et il restera neuf euros) ; ce dernier nombre (819) est une unité de plus que le quotient entier de m par b . D'où le code :

```
1 b= 10
2 m=8181
3 N=m//b
4 r=m%b
5 if r!=0:
6     N=N +1
7 print("b = ", b, "m =", m, "->", N)
8 print("->", N)
9
10 print()
11
12 b= 10
13 m=800
14 N=m//b
15 r=m%b
16 if r!=0:
17     N=N +1
18 print("b = ", b, "m =", m, "->", N)
19 print("->", N)
20
21 b = 10 m = 8181 -> 819
22 -> 819
23
24 b = 10 m = 800 -> 80
25 -> 80
```

- Lignes 16 et 17 : s'il est possible de payer *exactement* la somme m avec des billets b alors la condition ligne 16 est fausse et c'est N qui donne la réponse attendue.

Le plus grand, le plus petit

On donne deux entiers a et b . Ecrire un code qui calcule le plus petit et le plus grand des deux entiers a et b . Les valeurs calculées seront placées dans des variables `mini` et `maxi`.

Solution

Il suffit de comparer les deux nombres pour savoir qui est `mini` et qui est `maxi`, d'où le code :

```
1 a = 42
2 b = 17
3
4 if a < b:
5     mini=a
6     maxi=b
7 else:
8     mini=b
9     maxi=a
10
11 print(a, b)
12 print("mini =", mini, "maxi =", maxi)
```

```
13 42 17
14 mini = 17 maxi = 42
```

On pouvait aussi écrire le code suivant qui évite le `else` :

```
1 a = 42
2 b = 17
3
4 maxi=a
5 mini=b
6
7 if a < b:
8     mini=a
9     maxi=b
10
11 print(a, b)
12 print("mini =", mini, "maxi =", maxi)
```

EXERCICES

Nombre intermédiaire

On donne trois entiers a , b et c , par exemple $a = 42$, $b = 100$ et $c = 10$. On demande de déterminer et d'afficher le nombre qui est encadré par les deux autres. Dans l'exemple précédent, on a $c \leq a \leq b$ donc le nombre demandé est $a = 42$.

Transport en bus

Un bus peut contenir p passagers. Combien faut-il de bus pour transporter n passagers ? Coder en Python la réponse à cette question et appliquer à différents exemples (attention, la réponse dépend d'une condition laquelle, bien sûr, doit être exprimée à l'aide d'une instruction `if`).

Excédent payé

Soit à régler un montant de m euros avec des billets de 20 euros. Par exemple,

- si $m = 85$ alors il faut 5 billets de 20 euros
- si $m = 120$ alors il faut 6 billets de 20 euros

On demande de calculer le nombre `exc` représentant l'excédent payé en fonction de m . Par exemple,

- si $m = 85$ alors `exc` = 15.
- si $m = 120$ alors `exc` = 0.

Jeu à deux nombres

Dans cet exercice, on appellera *combinaison* la donnée de deux entiers a et b . Ainsi, on peut parler de :

- la combinaison 42, 17 ;
- la combinaison 81, 81.

À partir d'une combinaison, on a les règles suivantes :

- Si $a=b$, la combinaison rapporte 10 points.
- Si a et b sont consécutifs (comme $a=5$ et $b=4$), alors la combinaison rapporte 3 points.
- sinon la combinaison ne rapporte rien.

Écrire un code Python qui, étant donné une combinaison de deux entiers a et b , affiche le nombre de points que rapporte la combinaison.

Heures d'ouverture

Dans l'exercice, on supposera que

- les jours de la semaine sont codés à l'aide d'un entier j avec $j = 1$ pour lundi, $j = 2$ pour mardi, et ainsi de suite jusqu'à dimanche ($j = 7$).
- une heure est codée par un entier entre 0 (inclus) et 24 (exclu).

Un magasin est ouvert du lundi au vendredi de 8h à 18h et le samedi de 9h à 12h et il est fermé le reste du temps.

On demande d'écrire une fonction `f` prenant en paramètres un jour j et une heure h et qui renvoie `True` si le magasin est ouvert le jour j à l'heure h et `False` sinon.

Voici quelques exemples de comportements de la fonction `f` :

- `f(2,17)` vaut `True`
- `f(6,10)` vaut `True`
- `f(5,18)` vaut `True`
- `f(2,19)` vaut `False`
- `f(6,17)` vaut `False`

Arrondir l'heure

- ① On donne un entier $n \geq 0$. Construire une variable N valant le multiple de 5 le plus proche de n . Vous raisonnerez en fonction la valeur du reste de la division de n par 5.

Par exemple :

- si $n = 42$ alors $N = 40$

- si $n = 15$ alors $N = 15$
- si $n = 64$ alors $N = 65$
- si $n = 90$ alors $N = 90$
- si $n = 0$ alors $N = 0$

- ② Cette question fait appel à la question précédente. Si vous n'avez pas réussi à calculer m , vous pourrez utiliser le code ci-dessous pour le multiple de 5 le plus proche de l'entier n :

```
1 m = round(n/5)*5
```

Vous allez devoir écrire un code qui arrondit une heure donnée aux 5 minutes les plus proches. Une heure de la journée sera codée par deux nombres entiers h et m , où h est le nombre d'heures ($0 \leq h < 24$) et m le nombre de minutes ($0 \leq m < 60$). Par exemple, 14 h 05 est codée par $h = 14$ et $m = 5$ ou encore 4 h sera codée par $h = 4$ et $m = 0$.

On se donne une heure de la journée, représentée par deux nombres entiers h et m . Écrire un code qui calcule l'heure arrondie à 5 minutes près. Plus précisément, votre code devra construire deux variables H et M correspondant à l'heure arrondie. Voici quelques exemples de comportements attendus :

```
1 14h 53m -> [14, 55]
2 18h 31m -> [18, 30]
3 02h 10m -> [2, 10]
4 01h 02m -> [1, 0]
5 09h 58m -> [10, 0]
6 23h 58m -> [0, 0]
7 23h 57m -> [23, 55]
```

On fera attention de ne pas écrire des nombres de minutes ou d'heures avec un zéro initial (comme 02), Python 3 considérant cette syntaxe comme une erreur.

Équation du second degré

L'exercice consiste à écrire un code permettant de résoudre dans l'ensemble des réels une équation du second degré $ax^2 + bx + c = 0$. On supposera que a, b, c sont des entiers, que a est non nul et que l'inconnue x est un nombre réel.

Rappels : résolution de l'équation du second degré

On rappelle la résolution de l'équation. On calcule $\Delta = b^2 - 4ac$. Il y a alors trois cas :

- 1^{er} cas : $\Delta > 0$. L'équation admet deux solutions réelles, données par les formules

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a} \quad \text{et} \quad x_2 = \frac{-b + \sqrt{\Delta}}{2a}.$$

- 2^e cas : $\Delta = 0$. L'équation admet une seule solution réelle, $x = \frac{-b}{2a}$, donnée par n'importe laquelle des deux formules du 1^{er} cas.
- 3^e cas : $\Delta < 0$. L'équation n'admet aucune solution.

- ① Écrire un code Python qui détermine le nombre de solutions d'une équation du second degré.
- ② Affiner le code précédent pour résoudre complètement dans les réels l'équation $ax^2 + bx + c = 0$ et afficher les solutions éventuelles sous forme de nombres flottants (et pas de fractions). Tester les trois équations suivantes :

Equation	Solutions
$6x^2 - 5x + 1 = 0$	$1/2$ et $1/3$
$4x^2 - 12x + 9 = 0$	$3/2$
$6x^2 + 7x + 7 = 0$	Aucune

Chapitre III

Boucles for, listes, boucles while

1 Boucles for

a) Boucle `for` : introduction

Le code suivant présente un exemple d'utilisation d'une boucle `for` :

`for.py`

```
1 for i in range(0, 4):  
2     print("Bonjour !")  
3     print("-----")
```

```
4 Bonjour !  
5 -----  
6 Bonjour !  
7 -----  
8 Bonjour !  
9 -----  
10 Bonjour !  
11 -----
```

Les lignes 1-3 constituent une instruction `for`. On observe (lignes 4-11) que le message **Bonjour** ! suivi du séparateur `-----` est affiché 4 fois. Si on changeait la valeur 4 à la ligne 1 en la valeur 5 le message serait répété 5 fois.

Ici, la boucle `for` a permis de répéter une action un certain nombre de fois.

Déroulement d'une instruction `for`

- À la ligne 1, la présence de `range(0, 4)` permet d'itérer sur les entiers de 0 à 4, l'entier 4 étant exclu : 0, 1, 2 et 3. Les entiers vont être générés les uns à la suite des autres.
- Quand la boucle `for` commence, l'indice `i` vaut le premier élément généré (ici 0).
- L'exécution entre ensuite dans la ligne 2, effectue l'action d'affichage puis idem à la ligne 3
- Une fois les deux affichages effectués, l'exécution revient à la ligne 1. Comme pour l'instant `i = 0`, il reste encore des entiers à générer, donc, le principe même de l'instruction `for` veut que `i` passe à l'élément suivant, ie `i = 1` puis les instructions d'affichage des lignes 2-3 s'exécutent à nouveau.
- L'exécution se poursuit ainsi jusqu'à ce que `i` passe de 2 à 3. Lorsque `i` vaut 3, un affichage se fait encore et lorsque l'exécution repasse ligne 1, tous les entiers ont été générés : la conséquence est que la boucle s'arrête.

Édition d'une boucle `for`

On reprend le code `for.py`, on détaille la syntaxe d'une boucle `for` et le vocabulaire associé :

`for.py`

```
1 for i in range(0, 4):
2     print("Bonjour !")
3     print("-----")
```

- Ligne 1 : `for` et `in` sont des mots-clés de Python et sont obligatoires à toute boucle `for`.
- Ligne 1 : `i` est la *variable de contrôle* de la boucle `for`. D'autres noms fréquents de variables sont `j`, `k` mais tout nom de variable comme `toto` conviendrait. Ici, la variable de contrôle parcourt la succession des entiers générés par `range(0,4)`.
- Ligne 1 : le séparateur *deux-points* (`:`) à la fin est obligatoire
- Ligne 1 : `range` est une fonction built-in du langage Python qui sert à générer des entiers consécutifs.
- Ligne 1 : la partie qui commence à `for` jusqu'aux deux-points est appelé l'*en-tête* de la boucle `for`
- Lignes 2-3 : le *corps* de la boucle. Les instructions du corps de la boucle `for` sont celles qui vont être répétées tant que `i` est généré. Ce n'est pas le cas ici mais en général, le corps de la boucle dépend de la variable `i`. Le corps de la boucle `for` est dans l'immense majorité des cas, situé une ligne *sous* l'en-tête, autrement dit un saut de ligne dans le code-source est effectué après l'en-tête.
- Lignes 2-3 : le corps de la boucle `for` est toujours indenté et suit la règle générale d'indentation en Python.
- Lignes 1-3 : les trois lignes constituent une seule et même **instruction**, dite « instruction `for` ».

Le terme de *boucle* vient du fait que lorsque l'exécution est parvenue à la fin du corps de la boucle, l'exécution **retourne** à l'en-tête de la boucle.

Action qui dépend de la variable de la boucle

Dans le code `for.py`, l'action effectuée à chaque étape du parcours de la liste **ne** dépendait **pas** de la variable `i`. En fait, assez souvent, l'action effectuée dans le corps de la boucle **dépendra** de la variable de contrôle. Voici un exemple :

```
1 for i in range(0, 5):
2     print("Message", i+1, ":", "Bonjour !")
```

qui affiche

```
1 Message 1 : Bonjour !
2 Message 2 : Bonjour !
3 Message 3 : Bonjour !
4 Message 4 : Bonjour !
5 Message 5 : Bonjour !
```

On observe que chaque message affiché est différent du précédent par le numéro qu'il affiche : en effet, le corps de la boucle (ligne 2) dépend de `i` qui varie à chaque tour de boucle.

b) Répéter une action n fois avec une boucle **for**

La boucle **for** et la fonction `range` permettent typiquement de répéter une action n fois où n est fixé avant le démarrage de la boucle.

Par exemple, soit à afficher un carré de côté 10 comme ci-dessous :

```
1 * * * * *
2 * * * * *
3 * * * * *
4 * * * * *
5 * * * * *
6 * * * * *
7 * * * * *
8 * * * * *
9 * * * * *
10 * * * * *
```

L’affichage consiste en la répétition 10 fois de l’affichage d’une ligne de la forme

```
1 * * * * *
```

où les étoiles sont séparés par une espace.

Le code Python pourrait être :

```
1 z="* * * * *
2 for i in range(10):
3     print(z)
```

De même, soit à afficher les 10 premiers multiples de 5 en commençant par 75 :

75, $75 + 5 = 80$, $75 + 10 = 85$, etc.

Donc, un code répondant à la question est le suivant :

```
1 d = 75
2 for i in range(10):
3     print(d + 5 * i)
```

```
4 75
5 80
6 85
7 90
8 95
9 100
10 105
11 110
12 115
13 120
```

— Ligne 2 : typiquement, on utilise `range(10)` pour exprimer qu’une action va être exécutée 10 fois. Le nombre de répétitions (ici 10) est connu avant d’effectuer la boucle **for**.

— Ligne 3 : l’action (d’afficher le multiple) est à effectuer 10 fois.

Règle : Si un programme *répète* une action et si le nombre de répétitions à effectuer est connu à l’avance, le programme utilise une boucle **for** avec la syntaxe suivante :

```

1 for i in range(n):
2     # action a coder ICI

```

c) Affichage sur une même ligne et boucle for

Soit à réaliser avec une boucle for l’affichage suivant :

```

1 -----
2 0 1 2 3 4 5 6 7 8 9
3 -----

```

Pour éviter le saut de ligne après l’affichage de chaque entier et pour séparer chaque nombre du précédent d’une espace, on utilise l’argument nommé `end = " "` :

```

1 print("-----")
2 for i in range(10):
3     print(i, end = " ")
4 print()
5 print("-----")

```

```

6 -----
7 0 1 2 3 4 5 6 7 8 9
8 -----

```

- Ligne 7 : chaque entier est séparé du précédent par un espace. Aucun saut de ligne n’est effectué lors de l’exécution de la boucle.
- Ligne 4 : permet d’effectuer un saut de ligne entre le dernier chiffre affiché (ligne 7) et la ligne formée de pointillés (ligne 8)

Omettre l’appel à `print` en sortie de boucle provoque un affichage incorrect :

```

1 print("-----")
2 for i in range(10):
3     print(i, end = " ")
4 print("-----")

```

```

5 -----
6 0 1 2 3 4 5 6 7 8 9 -----

```

Cette technique s’applique aussi à des boucles `while`.

d) Itérer sur des entiers consécutifs

La fonction built-in `range` permet de générer automatiquement des entiers consécutifs, comme les entiers 6, 7, 8 et 9 :

```

1 r=range(6, 10)
2 for i in r:
3     print(i)

```

```

4 6
5 7
6 8
7 9

```

En pratique, on n’utilise pas la variable intermédiaire `r` ci-dessus et on écrit plutôt :

```

1 for i in range(6, 10):
2     print(i)

```

Bien observer que le deuxième argument de la fonction `range` (dans l'exemple, c'est 10) est exclu des nombres générés. Toutefois, le nombre d'entiers générés par `range(a, b)` est $b - a$ (en supposant que $a \leq b$).

Si $n \geq 0$ est un entier, l'expression `range(n)` est un raccourci syntaxique pour `range(0, n)` :

```

1 for i in range(3):
2     print(i)

```

```

3 0
4 1
5 2

```

EXERCICES

Répéter 2020

Afficher les uns en dessous des autres 10 fois de suite l'entier 2020.

Consécutifs

Afficher les uns en dessous des autres tous les entiers consécutifs entre 2020 et 2038.

Afficher deux listes l'une après l'autre

Écrire un code, utilisant deux boucles `for` *successives* qui produise l'affichage exact suivant (noter qu'il y a deux lignes dans le même affichage) :

```

1 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61
2 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84

```

Suite d'entiers

On considère la suite de nombres entiers dont les premiers termes sont :

```

1 10, 21, 43, 87, 175, 351, etc

```

Cette suite est construite de la manière suivante :

- son premier terme est 10
- si x est un terme de la suite, le terme suivant vaut $2x + 1$.

Par exemple, si $x = 87$ alors le terme suivant dans la suite est $2x + 1 = 2 \times 87 + 1 = 175$.

Écrire un code qui à partir d'un entier $n > 0$ affiche le n^{e} terme de la suite. Exemple de comportements selon la valeur de n :

```

1 3 -> 43
2 1 -> 10
3 7 -> 703

```

2 Listes

a) Notion de liste

En Python, une liste est une structure de données modifiable représentant une « succession » d'objets :


```

1 L = [2020, 42, 2038]
2 print(L)
3 [2020, 42, 2038]

```

- Ligne 1 : L désigne une liste, formée de trois éléments. Une paire de crochets entoure les éléments de la liste et ces éléments sont séparés par des virgules.
- Ligne 2 : La fonction `print` est le moyen le plus simple d'obtenir un affichage sommaire d'une liste L.

La liste est la structure couramment utilisée pour représenter des « listes » au sens usuel du terme, comme une liste de mots :

```

1 L = ["Printemps", "Été", "Automne", "Hiver"]
2 print(L)
3 ['Printemps', 'Été', 'Automne', 'Hiver']

```

Les éléments d'une liste peuvent aussi être des expressions utilisant des variables :

```

1 x=4
2 L = [2020, 10 * x + 2, 2038]
3 print(L)
4 [2020, 42, 2038]

```

Les listes ci-dessus sont des listes dites *littérales* : elles sont définies en explicitant ses éléments dans le code-source, placés entre crochets, un par un et en les séparant par des virgules.

b) Indexation des éléments d'une liste

Une propriété fondamentale des listes est qu'on accède directement à chaque élément d'une liste par un indice entier. Les différents éléments de la liste sont indexés par les entiers 0, 1, 2, etc. jusqu'à la fin de la liste.

```

1 L = [2020, 42, 2038]
2 print(L[0])
3 print(L[1])
4 print(L[2])
5 2020
6 42
7 2038

```

- Ligne 2 : les éléments de la liste sont indexés, **en commençant par 0**.
- Lignes 2-4 : on peut accéder à chaque élément de la liste L par son indice entouré d'une paire de crochets.

Attention ! l'indexation commence à 0 et non à 1.

Si une liste contient n éléments, ses indices sont tous les entiers entre 0 et $n - 1$, bornes 0 et $n - 1$ incluses, par exemple, une liste de 4 éléments est indexée par les entiers 0, 1, 2 et 3. L'élément d'indice i de la liste L est l'objet `L[i]`. L'indice 0 correspond toujours au premier élément de la liste et l'indice $n - 1$ d'une liste L de longueur n correspond toujours au dernier élément de L.

Tenter d'accéder à un indice se référant à un élément hors de la liste déclenche une erreur de type **IndexError** :

```
1 L = [2020, 42, 2038]
2 print(L[3])

3 Traceback (most recent call last):
4   File "liste_indice_impossible.py", line 2, in <module>
5     print(L[3])
6 IndexError: list index out of range
```

- Ligne 1 : les indices de la liste L sont : 0, 1 et 2
- Ligne 2 : on essaye d'accéder à un indice de L qui n'existe pas

La tentative d'accès à un élément d'une liste par un indice entier impossible est appelé un *débordement d'indice* ou aussi *dépassement d'indice*.

c) Opérations sur les éléments d'une liste

Les éléments d'une liste L sont de la forme L[i]. Cette notation peut être utilisée pour des opérations comme on le ferait avec des variables :

```
1 t = [5, 3, 25, 3]
2 x = t[0] * t[1] + t[2] * t[3]
3 print(x)

4 90
```

Dans une certaine mesure, on peut considérer une liste comme une suite ordonnée de variables.

d) Nombre d'éléments d'une liste

La fonction built-in **len** appliquée à une liste renvoie le nombre d'éléments de la liste :

```
1 t = [81, 12, 65, 31]
2 n = len(t)
3 print(n)

4 4
```

- Ligne 2 : n est le nombre d'éléments de la liste t.

Une liste peut contenir un nombre quelconque d'éléments ; en particulier une liste peut contenir un seul élément et même aucun élément. La liste littérale [] ne contient aucun élément :

```
1 L= [42]
2 print(len(L))
3 L= []
4 print(len(L))

5 1
6 0
```

- Ligne 1 : liste ayant un seul élément.
- Ligne 3 : la liste vide de longueur nulle (ligne 6).

e) Appartenance à une liste

L'opérateur `in` permet de tester l'appartenance d'un élément à une liste :

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2 print(75 in L)
3 print(81 in L)
```

```
4 False
```

```
5 True
```

L'opérateur `in` renvoie un booléen. Le mot `in` est un mot-clé du langage Python.

Pour tester l'appartenance de `a` à une liste `L`, Python parcourt toute la liste depuis le début (l'indice 0) et jusqu'à ce que l'élément `a` soit trouvé ou que la fin de `L` soit rencontrée. Cela signifie que le test d'appartenance, bien que tenant sur une seule instruction, peut être très coûteux si `L` est longue.

f) Non-appartenance à une liste

L'opérateur `not in` permet de tester la non-appartenance d'un élément à une liste :

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2 print(75 not in L)
3 print(81 not in L)
```

```
4 True
```

```
5 False
```

L'opérateur `not in` renvoie un booléen. L'expression `a not in L` a même valeur de vérité que `not a in L` :

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2 print(75 not in L)
3 print(not 75 in L)
```

```
4 True
```

```
5 True
```

Le test de non-appartenance est basé sur le même principe que le test d'appartenance : un parcours de la liste depuis son début et arrêt du parcours lorsque la réponse est connue.

g) Modifier les éléments d'une liste

C'est une propriété fondamentale des listes : il est possible d'en modifier les éléments. Pour cela :

- on accède à l'élément par son indice
- on modifie l'élément par affectation.

Voici un exemple

```
1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3
4 L[4] = 75
5 print(L)
```

```

6 [65, 31, 9, 32, 81, 82, 46, 12]
7 [65, 31, 9, 32, 75, 82, 46, 12]

```

- Ligne 4 : modification d'un élément de la liste par affectation ;
- Ligne 7 : comparer, la liste L a été modifiée.

Le fait de pouvoir modifier le contenu d'une liste se traduit en disant qu'une liste est un objet *mutable*. Si L est une liste, on peut non seulement changer ses éléments, mais lui en adjoindre, ou lui en supprimer et donc modifier le nombre d'éléments de la liste.

Comme pour une variable, on peut affecter ou réaffecter un élément d'une liste :

```

1 t = [5, 3, 25, 3]
2 t[0] = 10 * t[0]
3 print(t)
4 [50, 3, 25, 3]

```

- Ligne 2 : réaffectation de l'élément t[0].

h) **append : adjoindre un élément à une liste**

La méthode **append** modifie une liste en lui ajoutant un élément à sa fin. Exemple :

```

1 L = [65, 31, 9]
2 print(L)
3 L.append(81)
4 print(L)
5 [65, 31, 9]
6 [65, 31, 9, 81]

```

- Ligne 3 : l'élément 81 est ajouté à la fin de la liste.
- Lignes 4 et 6 : bien qu'ayant changée, la liste est toujours référencée par L.

On notera la syntaxe particulière de **append** utilisant un point (« la notation suffixe ») : chaque liste L admet un « attribut » **append** que l'on obtient par L.**append** et qui s'utilise comme une fonction.

La méthode **append** ne renvoie rien :

```

1 L = [65, 31, 9]
2 print(L)
3 x = L.append(81)
4 print(L)
5 print(x)
6 [65, 31, 9]
7 [65, 31, 9, 81]
8 None

```

- Ligne 3 : le membre de droite est un *appel* de la méthode **append**. Le membre de gauche référence le retour de cet appel.
- Ligne 8 : on observe qu'une fois l'appel à **append** effectué, x vaut **None** autrement dit *rien*.

Ainsi, contrairement à ce que l'on pourrait imaginer, L.**append**(truc) **ne** représente **pas** la liste L complétée par **truc**.

Adjoindre plusieurs éléments

Un unique appel à la méthode `append` ne permet pas d'ajouter plusieurs éléments. Pour ajouter plusieurs éléments, il faut appeler autant de fois la méthode `append` :

```
1 L = [65, 31, 9]
2 L.append(81)
3 L.append(100)
4 print(L)
5 [65, 31, 9, 81, 100]
```

i) Nature des éléments d'une liste

Les éléments d'une même liste peuvent ne pas être du même type et un objet de type quelconque peut être élément d'une liste :

```
1 L = [2020, 3.14, "ROSE"]
2 print(L)
3 [2020, 3.14, 'ROSE']
```

En réalité, les experts Python considèrent qu'une liste est faite pour collecter des données de même type, cf. l'unité 1 (§Tuple et liste : la différence selon BDFL§).

EXERCICE TYPE

Éléments central ou centraux

On se donne une liste d'entiers `L`. Afficher

- l'élément situé au milieu de la liste si la liste contient un nombre impair d'éléments
- les deux éléments centraux de la liste si la liste contient un nombre pair d'éléments.

Par exemple :

Liste	Éléments centraux
[31, 12, 81, 9, 65]	81
[31, 12, 81, 9, 65, 32]	81 9
[31, 12]	31 12
[81]	81

Solution

Si la liste contient un nombre impair n d'entiers, par exemple la liste suivante pour $n = 15$:

```
1 31 12 81 99 65 44 58 63 42 21 33 16 14 81 50
```

alors il y a un unique élément central. Quel est son indice i ? Dans notre exemple, c'est l'élément qui vaut 63 et son indice est 7. Dans le cas général, comme

$$n = k + 1 + k$$

on voit bien que l'élément central est le $k + 1$ -ème élément de la liste et donc son indice i est 1 de moins que son rang c'est-à-dire $i = k$.

Comme $n = 2k + 1$, on a $n//2 = k$ donc l'élément central de L est $L[n//2]$.

Si la liste contient un nombre pair n d'entiers, par exemple la liste suivante pour $n = 16$:

```
1 31 12 81 99 65 44 58 63 42 21 33 16 14 81 50 96
```

alors il y a deux éléments centraux. Quel sont leurs indices ? Dans notre exemple, ce sont les éléments qui valent 63 et 42 donc d'indice 7 et 8. Dans le cas général, comme

$n = k + k$

on voit bien que les éléments centraux sont aux rangs k et $k + 1$ et donc d'indices $k - 1$ et k . Or k représente $n//2$ et donc les éléments centraux sont

$L[n//2 - 1]$ et $L[n//2]$.

D'où le code :

```
1 L = [31, 12]
2 L=[81]
3 L = [31, 12, 81, 9, 65]
4 L = [31, 12, 81, 9, 65, 32]
5
6 n=len(L)
7 if n%2==1:
8     print(L[n//2])
9 else :
10    print(L[n//2-1], L[n//2])
11 81 9
```

EXERCICES

Liste : premier et dernier

On se donne une liste d'entiers L . Afficher sur une même ligne et séparés par une espace le premier élément de la liste et le dernier élément de la liste. Par exemple,

— si $L = [31, 12, 81, 9, 65]$ alors l'affichage est : 31 65

— si $L = [42]$ alors l'affichage est : 42 42

Liste : k -ème élément

On se donne une liste d'entiers L . On suppose que la liste L contient au moins deux éléments. On donne un entier $k \geq 1$. Afficher le k -ème élément de la liste si cet élément existe sinon, on affiche le message *rang invalide*.

Exemples d'exécution :

— $L = [31, 12, 81, 9, 65], k = 3 \rightarrow 81$

— $L = [31, 12, 81, 9, 65], k = 42 \rightarrow \text{rang invalide}$

Liste de trois éléments

On donne un entier x et on compte de 10 en 10 trois fois à partir de x . Construire la liste L qui contient les trois nombres obtenus. Par exemple, si $x=42$, la liste est formée des trois entiers 42, 52 et 62.

Changer certains éléments d'une liste

On donne une liste L d'entiers ayant au moins 3 valeurs. Modifier cette liste pour que les éléments x de la liste qui sont

- en première position
- en deuxième position
- en dernière position

soient remplacés par $10x$. Par exemple, si $L = [31, 12, 81, 9, 65]$ alors, après modification, on aura $L = [310, 120, 81, 9, 650]$.

Test de présence

On donne

- une liste d'entiers L
- un entier x

On rappelle que x **in** L est un booléen qui teste la présence de x dans la liste L .

Construire un booléen `ok` qui vaut **True** si l'un des trois entiers x , $x - 1$ et $x + 1$ est dans la liste et **False** sinon. Exemple avec $x = 42$:

- si $L = [31, 12, 42, 9, 65]$ alors `ok = True`
- si $L = [31, 12, 81, 43]$ alors `ok = True`
- si $L = [41]$ alors `ok = True`
- si $L = [31, 12, 81, 9, 65]$ alors `ok = False`

Liste des solutions de l'équation du second degré

Soit l'équation du second degré $ax^2 + bx + c = 0$. On supposera que a, b, c sont des entiers, que a est non nul et que l'inconnue x est un nombre réel.

On rappelle le code Python de la résolution d'une équation du second degré :

```
1 a = 2
2 b = -3
3 c = 1
4 delta = b*b -4*a*c
5 if delta >0:
6     print("2 solutions distinctes :")
7     print("x1 =", (-b - delta**0.5)/(2.*a))
8     print("x2 =", (-b + delta**0.5)/(2.*a))
9 else:
10     if delta == 0:
11         print("une seule solution")
12         print("x =", (-b)/(2.*a))
13     else:
14         print("Aucune solution")
15 # output
16 2 solutions distinctes :
17 x1 = 0.5
18 x2 = 1.0
```

Adapter le code précédente pour construire la liste S contenant toutes les solutions de l'équation. Par exemple,

- si $a = 2$, $b = -3$ et $c = 1$ alors $S = [1, 0.5]$
- si $a = 4$, $b = 4$ et $c = 1$ alors $S = [-0.5]$
- si $a = 4$, $b = 4$ et $c = 4$ alors S est la liste vide

Mois de 31 jours et liste

On donne un numéro de mois entre 1 et 12. Créer une variable booléenne `est_mois_31` (prenant comme valeur `True` ou `False`) qui teste si m est le numéro d'un mois ayant 31 jours¹ comme janvier (numéro 1) ou juillet (numéro 7) mais pas février (numéro 2). On utilisera impérativement l'appartenance à une liste

Tester les mois de 31 jours avec une liste

Construire une liste littérale `jours` telle que pour chaque numéro i entre 1 et 12 d'un mois d'une année non bissextile, `jours[i]` désigne le nombre de jours du mois de numéro i . Par exemple, `jours[10] = 31`.

En déduire une variable booléenne `mois31` qui, étant donné un numéro de mois i , vaut `True` si le mois de numéro i possède 31 jours et `False` sinon.

Echanger les extrémités d'une liste

Cette question nécessite d'adapter aux éléments d'une liste le code d'échange des valeurs de deux variables.

Soit une liste L . Modifier L pour que son premier élément et son dernier élément soient échangés. Par exemple, si $L = [31, 12, 9, 65, 81]$ alors après échange, on aura $L = [81, 12, 9, 65, 31]$.

Inverser une liste

Soit une liste L . Modifier L pour que

- le 1^{er} élément et le dernier élément soient échangés de position
- le 2^e élément et l'avant-dernier élément soient échangés de position
- ...
- et ainsi de suite jusqu'à ce que tous les échanges possibles soient effectués.

Par exemple,

- si $L = [31, 12, 9, 65]$ alors, après échanges, $L = [65, 9, 12, 31]$
- si $L = [31, 12, 81, 9, 65]$ alors, après échanges, $L = [65, 9, 81, 12, 31]$
- si $L = [31]$ alors, après échange, $L = [31]$

3 Parcourir, répéter

a) Boucle for : parcours d'une liste sans utiliser d'indices

Une boucle `for` peut parcourir n'importe quelle liste sans se référer aux indices des termes dans la liste.

Soit une liste d'entiers, par exemple

65, 31, 9, 32.

1. Les mois ayant 31 jours sont : janvier (1), mars (3), mai (5), juillet (7), août (8), octobre (10), décembre (12).

On veut afficher chaque entier x ainsi que $10 * x$, ce qui donnerait

```
1 65 650
2 31 310
3 9 90
4 32 320
```

On dispose donc d'une liste d'entiers $L=[65, 31, 9, 32]$. Il s'agit de **parcourir** L et d'effectuer une action à chaque élément de la liste. Ce type d'action répétée se code en Python à l'aide d'une « boucle for » portant sur les éléments de la liste.

Voici le code Python réalisant cet affichage :

```
1 L=[65, 31, 9, 32]
2
3 for x in L:
4     print(x, 10 * x )
```

L'interprétation est exactement la même que lorsque la boucle est appelée sur `range(n)` : la variable x parcourt la liste L et à chaque étape, le corps de la boucle (ligne 4) est exécuté.

b) Liste construite depuis la liste vide

Pour créer une liste d'éléments vérifiant une certaine propriété P , on procède souvent ainsi :

- on crée une liste L , initialement vide,
- on garnit successivement L d'éléments vérifiant P en utilisant la méthode `append`.

Exemple

Étant donné une liste t d'entiers, on veut extraire de t la liste L des entiers x tels que $x \geq 42$.

Solution :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 L= []
3 for i in range(len(t)):
4     if t[i] >= 42:
5         L.append(t[i])
6 print(L)
7 [65, 81, 82, 46]
```

- Ligne 2 : on crée une liste vide à laquelle on va adjoindre les éléments de L qui conviennent.
- Lignes 3-5 : on parcourt la liste t .
- Ligne 4 : on teste si l'élément courant vérifie la condition P .
- Lignes 4-5 : si un élément la vérifie, l'élément est placé à la fin de la liste L .

c) Boucle for et calcul de somme

On se donne une liste L d'entiers. On cherche à calculer la somme S des éléments de la liste L . Par exemple, si $L = [10, 3, 12, 5]$ alors $S = 10 + 3 + 12 + 5 = 30$.

On trouvera le code ci-dessous. L'idée de l'algorithme à implémenter est la suivante :

- on va faire évoluer une variable « accumulatrice » S au sein d'une boucle for qui parcourt la liste L par ses indices (lignes 4-7 ci-dessous) ;

- la variable `S`, en tout début de boucle, est initialisée à 0 (ligne 4) et en fin de boucle, elle vaut la somme attendue (lignes 9 et 10) ;
- à chaque étape `i` de la boucle, `S` est augmentée de l'élément courant `L[i]` de la liste `L` (ligne 7) : la variable `S` « accumule » ;
- `S` va prendre successivement les valeurs suivantes : 0, 10, 13, 25 et 30.

Voici un code traduisant exactement l'algorithme :

```

1 L = [10, 3, 12, 5]
2 n = len(L)
3
4 S = 0
5
6 for i in range(n):
7     S = S + L[i]
8
9 print(S)

```

```
10 30
```

La technique de variable accumulatrice s'emploie essentiellement dans des boucles. C'est une technique très fréquente d'emploi. La technique du *compteur* pour dénombrer le nombre d'éléments vérifiant une certaine propriété n'en est qu'un cas particulier. Un compteur est en général initialisé à 0.

d) Boucle for, filtrage et comptage

Étant donné une liste `L` d'entiers, soit à afficher les entiers pairs de `L` ; dans le jargon, on dit qu'on effectue un *filtrage* des éléments de `L`, le filtre étant ici le caractère pair de l'élément de la liste. Voici un code répondant au problème :

```

1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 for i in range(len(L)):
4     if L[i] % 2 == 0:
5         print(L[i])

```

```

6 32
7 82
8 46
9 12

```

- Lignes 3-5 : parcours de la liste
- Ligne 4 : filtrage avec une instruction `if`

Comptage

Pour un programme de comptage, on crée à l'aide d'une variable un *compteur*. La variable porte souvent le nom de `cpt` ou encore (en anglais) `cnt`. Le plus souvent la variable est initialisée à 0. Le comptage est souvent associé à un filtre qui indique ce que l'on compte. Par exemple, soit à compter les entiers pairs d'une liste donnée :

```

1 L = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 cpt=0
4
5 for i in range(len(L)):
6     if L[i] % 2 == 0:
7         cpt = cpt + 1
8
9 print(cpt)
10 4

```

- Ligne 3 : initialisation d'un compteur.
- Lignes 5-7 : parcours de la liste.
- Ligne 6 : filtrage.
- Ligne 7 : incrémentation du compteur lorsque l'élément est filtré.

e) Parcours complet d'une liste par indices

Comme on peut accéder à chaque élément d'une liste de n éléments par son indice i avec $0 \leq i < n$, un usage très fréquent² de la boucle **for** est le parcours d'une liste par ses indices.

Par exemple, soit une liste de nombres, comme `t = [65, 31, 9, 32, 81, 82, 46, 12]`. On cherche à afficher le produit $10 \times x$ pour chaque terme x de la liste :

```

1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2
3 for i in range(8):
4     print(10 * t[i])
5 650
6 310
7 90
8 320
9 810
10 820
11 460
12 120

```

- Ligne 1 : `t` contient 8 éléments donc pour parcourir `t` avec un indice `i`, on utilise l'en-tête `for i in range(8)`
- Ligne 4 : `t[i]` permet d'accéder à l'élément courant de `t`.

Code amélioré

Pour une meilleure lisibilité et une meilleure maintenance, un des deux codes ci-dessous est nettement préférable au code précédent :

2. Mais parfois considéré comme non pythonique, cf. [foreach loop](#) et [unpythonic loop](#).

```

1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 n = len(t)
3
4 for i in range(n):
5     print(10 * t[i])

```

```

6 t = [65, 31, 9, 32, 81, 82, 46, 12]
7
8 for i in range(len(t)):
9     print(10 * t[i])

```

- Ligne 2 ou ligne 8 : la longueur de `t` n'est pas écrite littéralement : si dans le code ligne 1, on modifie `t` en rajoutant un élément, le code en-dessous n'a pas besoin d'être modifié.

f) Parcours de liste : indices voisins de l'indice courant

L'avantage de parcourir une liste via ses indices est que l'on peut accéder à l'indice suivant ou l'indice précédent. Par exemple, soit à afficher toutes les sommes de deux termes successifs de la liste `t = [65, 31, 9, 32, 81, 82, 46, 12]`, à savoir les sommes suivantes :

96, 40, 41, etc.

L'algorithme de base consiste à parcourir `t` jusqu'à son avant-dernier terme, et à afficher $x + y$ où x est le terme courant de `t` et y le terme suivant dans `t` :

```

1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 n = len(t)
3
4 for i in range(n-1):
5     print(t[i] + t[i+1])

```

```

6 96
7 40
8 41
9 113
10 163
11 128
12 58

```

- Ligne 4 : Le choix de `(n-1)` se justifie par le fait que le parcours de `t` s'arrête à l'avant-dernier terme de `t` puisqu'il faut à chaque étape du parcours considérer le terme *suivant*.
- Ligne 5 : `t[i+1]` est le terme suivant du terme courant, ce qui suppose que $i + 1 < n$ et donc que $i < n - 1$
- Ligne 4 : écrire `range(n)` au lieu de `range(n-1)` entraînerait une erreur (un débordement d'indice) lorsque $i = n - 1$ et que l'interpréteur calcule `t[i+1]`.

g) Boucle for et calcul de maximum

La technique de la « variable accumulatrice » peut s'adapter au calcul du plus grand élément des termes d'une liste :

```

1 L = [10, 3, 12, 5]
2 maxi = L[0]
3 n = len(L)
4
5 for i in range(1, n):
6     if L[i] > maxi:
7         maxi = L[i]
8
9 print(maxi)
10 12

```

- Ligne 1 : on observe que le plus grand élément de L est 12.
- Ligne 2 : variable initialisée au premier terme de la liste et qui, en fin de programme, contiendra le maximum la liste L.
- Lignes 5-7 : la liste est parcourue et à la fin de chaque itération, `maxi` est le plus grand de tous les termes examinés.
- Ligne 5 : le `range` commence à 1 car comme `maxi` est initialisé à `L[0]`, il est inutile d'effectuer la comparaison pour `i = 0` puisque `L[0] > maxi` (cf. ligne 6) autrement dit `L[0] > L[0]` vaut `False`.

Même si ce n'est pas formellement interdit, on évitera d'appeler `max` la variable accumulatrice du maximum car cela empêcherait d'utiliser la fonction Python native nommé `max`. De même on évitera le nom `min` pour un minimum.

h) La technique du drapeau dans une boucle for

Soit à définir un code Python qui partant d'une liste L d'entiers détermine si L ne contient que des entiers positifs. Plus précisément, le code doit définir un booléen nommé `tousPositifs` qui vaut `True` si L ne contient que des entiers positifs et `False` sinon.

Le code suivant réalise cette tâche sur deux exemples de liste :

```

1 # 1er exemple
2 L= [31, 82, -42, 32, -81]
3
4 tousPositifs = True
5 for i in range(len(L)):
6     if L[i] < 0:
7         tousPositifs = False
8 print(L, '->', tousPositifs)
9
10 print("-----")
11
12 # 2e exemple
13 L= [31, 82, 421, 32, 81]
14
15 tousPositifs = True
16 for i in range(len(L)):
17     if L[i] < 0:
18         tousPositifs = False
19 print(L, '->', tousPositifs)

```

```

20 [31, 82, -42, 32, -81] -> False
21 -----
22 [31, 82, 421, 32, 81] -> True

```

- Le même code (lignes 4-8 et lignes 15-19) est testé avec deux listes différentes (lignes 2 et 13). Ci-dessous, les commentaires ne portent que sur le premier groupe de code.
- Dans chaque cas, on affiche la liste L (lignes 8) et le booléen `tousPositifs`.
- Lignes 4 : on définit au début du code un « drapeau », ici le booléen `tousPositifs`; ce drapeau témoigne d'un état, l'état étant que la liste L contienne uniquement des nombres positifs (`True`) ou non (`False`). Au départ, le drapeau `tousPositifs` est placé sur `True`.
- lignes 6-7 : La liste L est parcourue et si au cours du parcours de L, la présence dans la liste d'un entier strictement négatif est détectée (ligne 6), le drapeau change de couleur, ici il passe à `False` (ligne 7). Le drapeau ne change **que si** un entier négatif est détecté, autrement dit le `if` n'est pas couplé à un `else`.
- Ligne 5-8 : si le drapeau change par rapport à son état initial, c'est qu'un entier négatif a été détecté dans la liste L et donc il est attendu que le drapeau `tousPositifs` vaille `False`, ce qui est bien le cas, cf. ligne 20.
- Lignes 15-18 : si au contraire, la liste L est parcourue sans que le drapeau ait changé, c'est qu'aucun entier négatif n'est jamais rencontré et donc il est attendu que le drapeau `tousPositifs` vaille `True`, ce qui est bien le cas, cf. ligne 22.

Ainsi le code ci-dessus réalise bien la tâche attendue.

Initialisation du drapeau

Comment est déterminée la valeur initiale du drapeau ? Réponse : en fonction de ce que représente le booléen une fois toute la liste parcourue. Si le booléen `True` doit correspondre au fait que tous les entiers sont positifs, c'est que le booléen doit changer en `False` dans la boucle quand un entier négatif est détecté et donc, le booléen doit être initialisé à `True`. Il serait possible de réaliser le même objectif en utilisant un autre drapeau initialisé à `False` :

```

1 L= [31, 82, -42, 32, -81]
2
3 contientNegatif = False
4 for i in range(len(L)):
5     if L[i] < 0:
6         contientNegatif = True
7 print(L, '->', contientNegatif)
8
9 print("=====")
10
11 L= [31, 82, 421, 32, 81]
12
13 contientNegatif = False
14 for i in range(len(L)):
15     if L[i] < 0:
16         contientNegatif = True
17 print(L, '->', contientNegatif)

```

```

18 [31, 82, -42, 32, -81] -> True
19 =====
20 [31, 82, 421, 32, 81] -> False

```

- Ligne 3 : on définit un drapeau `contientNegatif` qui vaudra `True` si `L` contient au moins un entier strictement négatif et `False` sinon.
- Lignes 5-6 : lorsqu'on parcourt la liste, le drapeau doit passer à `True` si un entier strictement négatif apparaît, donc le drapeau doit être initialisé à `False`.

EXERCICES TYPE

De 42 en 42

Afficher les entiers obtenus en comptant 10 fois de 42 en 42 à partir de 421.

Solution

Il s'agit de répéter une action 10 fois, donc le code a la forme suivante :

```

1 for i in range(10):
2     # Code
3     # à compléter

```

Les nombres à afficher vont évoluer dans une variable `x` initialisée à 421. A chaque tour de boucle, `x` est incrémenté de 42. D'où le code :

```

1 x=421
2 for i in range(10):
3     print(x)
4     x = x + 42

```

```

5 421
6 463
7 505
8 547
9 589
10 631
11 673
12 715
13 757
14 799

```

Autre code possible :

```

1 for i in range(10):
2     print(421 + i * 42)

```

Somme des positifs

On donne une liste `L` d'entiers et on demande de calculer la somme `s` des éléments positifs de cette liste. Si aucun élément de la liste n'est positif, on conviendra que la somme `s` vaut 0.

Exemples

```
1 L = [-2, 6, 0, -1, 4, 5, -3] -> s = 15
2 L = [-2, -3] -> s = 0
3 L = [42] -> s = 42
```

Solution

Le principe de calcul est le suivant, il s'agit d'une légère variante du calcul d'une somme : on parcourt la liste et on ajoute à une variable accumulatrice `s` initialisée à 0 chaque terme de la liste qui est strictement positif :

```
1 L=[-2, 6, 0, -1, 4, 5, -3]
2 s=0
3
4 for i in range(len(L)):
5     if L[i] > 0:
6         s = s + L[i]
7
8 print(L, "->",s)
9 [-2, 6, 0, -1, 4, 5, -3] -> 15
```

Somme alternées plus/moins

On se donne un entier positif `n`, par exemple, `n = 7`. On cherche à calculer la « somme »

$$1 - 2 + 3 - 4 + 5 - \dots n$$

où on alterne tantôt une addition, tantôt une soustraction. Pour `n = 7`, la somme vaut

$$1 - 2 + 3 - 4 + 5 - 6 + 7 = 4$$

Ecrire un code qui calcule `s` à partir de `n`. On pourra vérifier que pour `n` valant 2020 la somme vaut -1010.

Solution

En examinant l'exemple donné, on comprend qu'on effectue une addition sur les entiers impairs et une soustraction sur les entiers pairs. Il suffit donc de parcourir tous les entiers `i` entre 1 et `n` et d'ajouter ou retrancher `i` à une variable accumulatrice `s` initialisée à 0 selon que l'entier courant `i` est impair ou pair :


```

1 n=2020
2 s=0
3
4 for i in range(1, n+1):
5     if i % 2 == 1:
6         s = s + i
7     else:
8         s = s - i
9
10 print(s)
11 -1010

```

On peut aussi donner une solution plus matheuse en remarquant que la somme cherchée n'est autre que

$$\sum_{i=1}^n (-1)^{i+1} i$$

ce qui donne le code suivant :

```

1 n=2020
2 s=0
3
4 for i in range(1, n+1):
5     s = s - i*(-1)**i
6
7 print(s)
8 -1010

```

Somme sauf des extrémités

On donne une liste L d'entiers. Calculer la somme s des éléments de L en excluant du calcul de la somme le premier et le dernier élément de L. Par exemple, si L = [310, 12, 8100, 90, 31] alors s = 8202.

Solution

La solution qui peut venir spontanément à l'esprit est de parcourir la liste avec l'idiome habituel et de filtrer avec une instruction **if** pour savoir si le terme courant est au début ainsi qu'à la fin et ainsi exclure ces termes de la somme. Ce qui donnerait le code suivant :

```

1 L = [310, 12, 8100, 90, 31]
2 n=len(L)
3
4 s=0
5 for i in range(n):
6     if i != 0 and i != n -1:
7         s=s+L[i]
8 print(s)

```

9 8202

C'est bien le résultat attendu mais le code est particulièrement maladroit. En effet, l'instruction **if** ne sert que deux fois sur toute la liste et à des endroits prévisibles. Or, le début de la liste est à l'indice 0 et la fin à l'indice $n-1$ où $n = \text{len}(L)$. Par suite, il suffit juste d'itérer entre les indices 1 et $n - 2$. D'où le code :

```
1 L = [310, 12, 8100, 90, 31]
2 s=0
3 for i in range(1,len(L)-1):
4     s=s+L[i]
5 print(s)
```

6 8202

Des mesures de vitesse d'exécution montreraient que le premier code à une pénalité de 10

Alternance de parité

On donne une liste L d'entiers et on demande de créer un booléen **alterneParite** valant **True** si les éléments de L se suivent en alternant de parité et **False** sinon. Voici des exemples de comportements attendus :

L	Alternance de parité
[81, 32, 9, 12]	True
[32, 9, 32, 65]	True
[32, 9, 31, 82]	False
[81]	True

Solution

Il s'agit d'appliquer la technique du drapeau. Il faut parcourir la liste de L. Mais comme il faut à chaque étape comparer les parités de deux éléments consécutifs, la boucle de parcours a la forme suivante :

```
1 n=len(L)
2
3 for i in range(n-1):
4     # Code à compléter
```

La parité d'un entier a est donnée par le reste $a \% 2$. Donc, deux éléments consécutifs de L ont des parités différentes si $L[i] \% 2 \neq L[i+1] \% 2$. Dans le cas contraire, le drapeau de surveillance de parité doit changer. D'où le code suivant :

```

1 L = [81, 32, 9, 12]
2 alterneParite = True
3
4 for i in range(0, len(L)-1):
5     if L[i] % 2 == L[i+1] % 2:
6         alterneParite = False
7
8 print(alterneParite)
9 True

```

Testons plusieurs listes placées dans une liste `tests` :

```

1 tests = [[81, 32, 9, 12], [32, 9, 32, 65], [32, 9, 31, 82], [81]]
2
3 for L in tests:
4
5     alterneParite = True
6
7     for i in range(0, len(L)-1):
8         if L[i] % 2 == L[i+1] % 2:
9             alterneParite = False
10
11     print(L, "->", alterneParite)

```

```

12 [81, 32, 9, 12] -> True
13 [32, 9, 32, 65] -> True
14 [32, 9, 31, 82] -> False
15 [81] -> True

```

On pourra remarquer que le code

```

1 L = [81, 32, 9, 12]
2 alterneParite = True
3
4 for i in range(0, len(L)-1):
5     if L[i] % 2 == L[i+1] % 2:
6         alterneParite = False
7
8 print(alterneParite)

```

n'est pas parfaitement optimal, déjà parce que la technique du drapeau n'est pas optimale et aussi parce que la parité de chaque élément de la liste ayant un voisin est évaluée **deux** fois (cf. ligne 5).

Afficher des nombres par paires

On se donne deux entiers a et b , avec $a \leq b$. Soit S la succession de tous les entiers depuis l'entier a inclus jusqu'à l'entier b inclus. On demande d'afficher tous les entiers de S par lignes de deux entiers consécutifs, sauf, **éventuellement**, la dernière ligne qui ne contiendra que le dernier élément de S . Le tableau ci-dessous donne quatre exemples :

a	b	Affichage
2	9	<pre> 1 2 3 2 4 5 3 6 7 4 8 9 </pre>
2	8	<pre> 1 2 3 2 4 5 3 6 7 4 8 </pre>
2	2	<pre> 1 2 </pre>
2	3	<pre> 1 2 3 </pre>

Bien tester tous ces cas.

Solution

On compte entre a et b un nombre de $N = b - a + 1$ entiers. Puisque les entiers doivent être groupés par deux il y aura $q = N // 2$ lignes de deux entiers et éventuellement une ligne de plus. Tout dépend de la parité de N . L'idée est donc d'afficher les q premières lignes et selon les cas de rajouter une ligne. D'où le code suivant :

```

1 a=2
2 b=9
3 N=b+1-a
4 q=N//2
5 x=a
6
7 for i in range(q):
8     print(x, x+1)
9     x=x+2
10 if N%2==1:
11     print(b)

```

```

12 2 3
13 4 5
14 6 7
15 8 9

```

- Lignes 5 et 9 : x est le nombre qui est affiché en chaque début de ligne (sauf peut-être la dernière).
- Ligne 9 : en effet, les nombres vont deux par deux.
- Lignes 11-12 : s'il y a un nombre impair de nombres à afficher, c'est que le dernier est seul sur sa ligne, et le dernier entier à afficher est b .

Le code ci-dessus ne montre qu'un seul exemple, pour en voir d'autres, changer les valeurs de a ou b et exécuter à nouveau le code.

On peut simplifier le code de la manière suivante :

```

1 a=2
2 b=6
3
4 for i in range((b+1-a)//2):
5     print(a, a+1)
6     a=a+2
7 if a-1!=b:
8     print(b)
9 2 3
10 4 5
11 6

```

- La variable `x` du précédent code n'est pas indispensable, il suffit d'utiliser `a` mais la valeur initiale de `a` est alors perdue à la fin du programme.
- Une fois que les lignes contenant deux entiers sont affichés (lignes 4-6) on regarde si le dernier entier affiché est `b`. Si c'est la cas, tous les entiers ont été affiché et il n'y a rien de plus à afficher. Si ce n'est pas le cas, il faut afficher `b` (ligne 8). Quel est le dernier nombre affiché? réponse : c'est $a - 2 + 1$ et donc $a - 1$, d'où la ligne 7.

Il était également possible de ne pas diviser par deux le nombres d'entiers à afficher :

```

1 a=2
2 b=8
3 cpt=0
4
5 for i in range(a,b+1):
6     cpt+=1
7     if cpt ==2:
8         print(i-1, i)
9         cpt=0
10
11 if cpt !=0:
12     print(b)
13 2 3
14 4 5
15 6 7
16 8

```

Le principe de ce code est de compter les nombres par groupes de 2 à l'aide du compteur `cpt` et lorsque le compteur atteint 2, les nombres sont affichés (ligne 8) et le compteur est remis à zéro (ligne 9).

Comptage de 5 en 5

On compte de 5 en 5 tous les entiers entre $a = 1914$ et $b = 2020$ (bornes incluses). À l'aide d'une boucle `for`, déterminer combien d'entiers on compte ainsi (on en trouvera 22).

Solution

Pour compter de 5 en 5 dans une suite d'entiers consécutifs, il suffit de les numéroter et de ne compter que les multiples de 5 dans l'énumération.

D'où le code :

```
1 a=1914
2 b=2020
3 cpt = 0
4 j=0
5
6 for i in range(a, b+1):
7     if j%5==0:
8         cpt = cpt + 1
9         j = j + 1
10
11 print(cpt)
12 22
```

- Ligne 6 : la variable `i` est factice, elle permet juste à la boucle `for` de s'exécuter.
- Lignes 4 et 9 : la variable `j` sert à énumérer les entiers parcourus.
- Ligne 8 : on compte de 5 en 5.

Liste des N premiers multiples de d

Soient N et d deux entiers positifs. Créer la liste L contenant les N premiers multiples de d en commençant par d . Par exemple, si $N = 7$ et $d = 10$ alors $L = [10, 20, 30, 40, 50, 60, 70]$.

Solution

```
1 N=7
2 d=10
3 L=[]
4 for i in range(1,N+1):
5     L.append(i*d)
6 print(L)
```

EXERCICES

Parcours *indice* -> *élément*

Etant donné une liste, afficher ligne par ligne chaque indice de L , suivi d'une flèche -> suivie de la valeur de L correspondant à l'indice. Par exemple, si $L = [31, 12, 81, 9, 31]$ alors l'affichage est

```
1 0 -> 31
2 1 -> 12
3 2 -> 81
4 3 -> 9
5 4 -> 31
```

Entiers consécutifs en décroissant

On donne deux entiers a et b avec $a \geq b$, par exemple $a = 2020$ et $b = 2000$. Afficher les entiers consécutifs depuis a jusqu'à b en ordre décroissant.

On observera qu'il y a $a - b + 1$ nombres à afficher et on pourra, par exemple, introduire dans une boucle for une variable auxiliaire j initialisée à a . Mais d'autres codes sont envisageables.

Modifier une liste en multipliant par 10

On donne une liste d'entiers. Modifier cette liste pour que chaque élément x de la liste soit remplacé par $10x$. Par exemple, si $L = [31, 12, 81, 9, 65]$ alors, après modification, on aura $L = [310, 120, 810, 90, 650]$.

Somme des n premiers entiers

On se donne un entier $n \geq 0$. Calculer la somme $1 + 2 + \dots + n$. Vérifier que le résultat obtenu est bien $\frac{n(n+1)}{2}$.

Compter de 0,5 en 0,5

On donne deux entiers a et b avec $a \leq b$ et on demande d'afficher tous les nombres entre a et b si on compte de 0,5 en 0,5. Par exemple si $a = 42$ et $b = 49$, il faut afficher les nombres suivants, de préférence sur une même ligne :

```
42 42.5 43 43.5 44 44.5 45 45.5 46 46.5 47 47.5 48 48.5 49
```

Il est attendu qu'un entier soit affiché sans point décimal, par exemple 42 et pas 42.0

Non multiples

Combien y-a-t-il d'entiers entre 1 et 20000 qui ne sont ni pairs ni multiples de 5 ? Par exemple, 421 ou 2017 font partie des entiers concernés mais pas 42 ni 2015. Le nombre à trouver est 8000.

Produit d'entiers

Calculer le produit P des entiers entre 42 et 421, autrement dit $P = 42 \times 43 \times \dots \times 420 \times 421$.

On trouvera un très grand entier qui commence par 1484.

Somme des multiples de 10

On donne une liste L d'entiers. Calculer la somme s des éléments de L qui sont des multiples de 10. Par exemple, si $L = [310, 12, 8100, 90, 31]$ alors $s = 8500$.

Somme suivant les indices impairs

On donne une liste L d'entiers. Calculer la somme s des éléments de L dont l'indice est impair. Par exemple, si $L = [31, 12, 81, 9, 31]$ alors $s = 21$.

Calcul du minimum

On donne une liste L d'entiers. Calculer le plus petit élément mini de la liste. Par exemple, si $L = [31, 9, 81, 9, 31]$ alors $\text{mini} = 9$.

Indice du minimum

On donne une liste d'entiers. Déterminer un indice i_mini correspondant au plus petit élément mini de la liste. Par exemple

— si $L = [31, 12, 81, 9, 31]$ alors $\text{mini} = 9$ et $i_mini = 3$

— si $L = [31, 9, 81, 9, 31]$ alors $\text{mini} = 9$ et $i_mini = 1$ (ou encore $i_mini = 3$)

Minimum des entiers d'indices pairs

On donne une liste L d'entiers. Calcul mini_pairs le plus petit des éléments d'indices pairs de la liste. Par exemple, si $L = [81, 32, 12, 9, 10, 65, 46]$ alors $\text{mini_pairs} = 10$.

Somme alternée plus/moins en décroissant

On cherche à écrire un code capable de calculer des expressions du genre

$$7 - 6 + 5 - 4 + 3 - 2 + 1.$$

Plus précisément, on se donne un entier $n > 0$ (ci-dessus, c'était $n = 7$) et on demande d'écrire un code qui renvoie la valeur de l'expression :

$$n - (n - 1) + (n - 2) - (n - 3) + \dots$$

expression qui se poursuit jusqu'à ce que le terme courant vaille 1. Noter que l'expression commence par n et alterne soustraction et addition. Si $n = 2037$ ou $n = 2038$ on trouvera que la somme vaut 1019.

Afficher par paires verticales

On donne deux entiers a et b et on demande d'afficher, par ordre croissant tous les entiers de a à b sur deux lignes en plaçant alternativement les entiers sur la ligne du haut et sur la ligne du bas. Par exemple, si $a = 2016$ et $b = 2023$ le code doit afficher

```
1 2016 2018 2020 2022
2 2017 2019 2021 2023
```

et si $a = 2020$ et $b = 2038$ le code doit afficher

```
1 2020 2022 2024 2026 2028 2030 2032 2034 2036 2038
2 2021 2023 2025 2027 2029 2031 2033 2035 2037
```

Valeurs paires d'une liste

Soit une liste L d'entiers. Séparer cette liste en deux listes :

- la liste P des entiers pairs de L
- la liste I des entiers impairs de L

Par exemple, si $L = [31, 12, 9, 65, 81, 42]$ alors $P = [12, 42]$ et $I = [31, 9, 65, 81]$

Indices pairs

Soit une liste L . Séparer cette liste en deux listes :

- la liste IP des éléments de L d'indices pairs
- la liste II des éléments de L d'indices impairs

Par exemple, si $L = [31, 12, 9, 65, 81, 42]$ alors $IP = [31, 9, 81]$ et $II = [12, 65, 42]$.

Décompte pair/impair

On donne une liste L d'entiers et on demande de calculer

- le nombre n_pair d'indices i tel que l'élément $L[i]$ est pair
- le nombre n_impair d'indices i tel que l'élément $L[i]$ est impair

Par exemple, si $L = [31, 12, 9, 65, 81, 42]$ alors $n_pair = 2$ et $n_impair = 4$.

Minimum des entiers pairs (boucle **for**)

On donne une liste L d'entiers contenant au moins un entier pair. Ecrire une fonction `miniPairs` qui renvoie le plus petit des éléments pairs de la liste. Exemples de comportements :


```

1 [81, 32, 12, 9, 12, 65, 46] -> 12
2 [81, 65, 46] -> 46

```

Minimum présent une seule fois

On donne une liste L d'entiers. Soit **mini** le plus petit élément de la liste. Ecrire une fonction **f** qui renvoie **True** ou **False** selon que **mini** apparaît une seule fois dans la liste ou au contraire, plusieurs fois. Par exemple

- si L = [31, 12, 81, 9, 31] alors **mini** = 9 et **f** renvoie **True** ;
- si L = [31, 9, 81, 9, 31] alors **mini** = 9 et **f** renvoie **False**.

Pièces de monnaie

Dans un pays imaginaire, les pièces de monnaie sont de **a** unités ou de **b** unités. On demande d'écrire un code qui détermine si, oui ou non, il est possible de régler un certain montant donné de **m** unités. Par exemple si **a** = 49 et **b** = 10 alors on ne peut pas régler **m** = 42 unités ni **m** = 105 mais on peut régler **m** = 128 puisque $128 = 3 \cdot 10 + 2 \cdot 49$.

Vous ne devez pas utiliser de boucles imbriquées.

Somme de n entiers consécutifs à partir de d

On vous donne deux entiers positifs **d** et **n**. On vous demande de calculer la somme des **n** entiers consécutifs à partir de **d**. Par exemple, si **d** = 10 et **n** = 4, vous devez calculer $10 + 11 + 12 + 13 = 46$. Voici d'autres exemples :

```

1 (d, n) = (10, 1) -> 10
2 (d, n) = (10, 2) -> 21
3 (d, n) = (10, 100) -> 5950

```

Le double du précédent

On donne une liste L d'entiers. Définir un booléen **ok** qui vaut **True** si chaque élément de la liste est le double du précédent et qui vaut **False** sinon. Exemple de comportements :

```

1 [10, 20, 40, 80] -> True
2 [12, 24, 48] -> True
3 [42] -> True
4 [0, 0, 0, 0, 0] -> True
5 [10, 40, 80] -> False

```

Liste constante

On donne une liste L et on demande de créer un booléen **tousEgaux** valant **True** si tous les éléments de la liste L sont égaux et **False** sinon. Par exemple si L = [42, 42, 42] alors **tousEgaux** vaudra **True** et si L = [42, 421, 42, 42] alors **tousEgaux** vaudra **False**.

Listes « opposées »

Ecrire un code qui partant deux listes d'entiers L et M crée un booléen **sontOpposees** valant **True** si les deux listes sont « opposées » et **False** sinon. Deux listes sont considérées comme « opposées » si elles ont le même nombre d'éléments et si, à des indices identiques, elles possèdent des éléments opposés (comme -81 et 81). Voici quelques exemples de comportements attendus :

```

1 [81, -12, 0, -81, -31] [-81, 12, 0, 81, 31] -> True
2                                     [-81] [81] -> True
3                                     [0, 0] [0, 0] -> True
4                                     [ ] [ ] -> True
5                                     [81, -12] [-81, -12] -> False
6                                     [-81, 12, 0] [81, -12] -> False

```

Suite croissante d'entiers consécutifs

Écrire un code qui à partir d'une liste L d'entiers définit une variable booléenne nommée **consecutifs** qui vaut **True** si la liste est constituée d'entiers CONSÉCUTIFS croissants et **False** sinon. Ci-dessous, voici quelques exemples de comportements attendus

```

1 [81, 82, 83] -> True
2 [82, 81, 83] -> False
3 [2013, 2038, 3000] -> False
4 [81] -> True

```

4 Les chaînes

a) Les chaînes : présentation

« Variables

La notion de chaîne

Les chaînes de caractères permettent de stocker du **texte**. En première approximation, une chaîne est une succession de caractères imprimables. Cela peut être, par exemple, une phrase ou un groupe de mots, comme *Le nombre 42 est magique*.

On dit souvent *chaîne* au lieu de *chaîne de caractères*.

De la même façon qu'un programme peut travailler avec des entiers, il peut travailler avec des chaînes.

Sensibilité à la casse

La *casse* d'un caractère alphabétique est le fait que le caractère soit en majuscule ou en minuscule.

Les chaînes *bonjour* et *bonJour* sont considérées en Python comme distinctes. Si deux chaînes ont, à des positions identiques, des lettres de casse différentes, les chaînes sont considérées comme distinctes.

Chaîne littérale

Le moyen le plus immédiat d'utiliser des chaînes dans un code Python est par le biais d'une *chaîne littérale* :

```

1 toto = "orange"
2 print(len(toto))

```

À la ligne 1, on lit **"orange"** : c'est une chaîne, dite *littérale*. Une chaîne est *littérale* lorsque

— tous ses caractères sont placés dans le code-source

— la chaîne est encadrée par des caractères particuliers, appelés *quotes*, ici des guillemets anglo-saxons.

Une chaîne littérale peut aussi être encadrée d’apostrophes :

```
1 toto = 'orange'
2 print(len(toto))
```

Afficher une chaîne

Pour afficher une chaîne avec une « commande » Python en mode fichier, on utilisera obligatoirement la fonction `print` :

```
1 toto = "orange"
2 print(toto)
3 print("bonjour")
4 orange
5 bonjour
```

- on affiche une chaîne par l’intermédiaire d’une variable
- Affichage direct d’une chaîne littérale.
- On notera que les délimiteurs de chaîne ne sont pas affichés.

Erreur courante de débutant

Pour afficher une chaîne littérale, on fera attention de ne pas oublier les quotes autour de la chaîne :

```
1 print(bonjour)
2 NameError: name 'bonjour' is not defined
```

b) La notion de chaîne littérale

Une chaîne littérale est un chaîne de caractères **saisie dans le code-source** entre une paire de délimiteurs. Ainsi, `"toto"` est une chaîne littérale, le délimiteur étant ici un guillemet :

```
1 s = "orange"
2 print(s)
3 orange
```

Les délimiteurs ne font pas partie du contenu d’une chaîne littérale.

Insistons : une chaîne littérale n’a de sens que dans un code-source puisque *littéral* veut dire qui s’interprète *littéralement* autrement dit comme cela s’écrit.

Littérale ou pas

Une chaîne n’est pas toujours une chaîne littérale. Par exemple `"Rose" + "Kiwi"` qui est une chaîne `s` :

```
1 s = "Rose" + "Kiwi"
2 print(s)
3 RoseKiwi
```

mais n'est pas exactement une chaîne **littérale** puisque `s` est le résultat d'une *opération* entre deux chaînes littérales.

Il existe bien un type chaîne (`str`) mais il n'existe pas de type « chaîne littérale ». Une chaîne littérale est juste une représentation « lettre à lettre »³ d'une chaîne dans un **code source-Python**.

Caractères unicodes

Par défaut, les caractères du contenu d'une chaîne littérale Python appartiennent au jeu Unicode et donc peuvent comporter des accents ou être des caractères spéciaux. Par exemple, la chaîne littérale `s` suivante est valide :

```
1 s = "Les pièces de 5€ n'existent pas"
2 print(s)
3 Les pièces de 5€ n'existent pas
```

Délimiteurs de chaînes littérales

Le délimiteur de chaîne littérale le plus familier est le guillemet anglais `"`. Un autre délimiteur usuel est l'apostrophe :

```
1 s = "toto"
2 t = 'toto'
3 print(s)
4 print(t)
5 toto
6 toto
```

- chaîne littérale entourée de guillemets
- chaîne littérale entourée d'apostrophes

Deux chaînes valides de même contenu mais encadrées par des délimiteurs différents sont égales :

```
1 print("toto" == 'toto')
2 True
```

c) Opérations sur des chaînes

On peut faire de nombreuses opérations sur les chaînes. Voici juste quelques exemples.

Nombre de caractères

Déterminer le nombre de caractères d'une chaîne, ce qu'on appelle sa *longueur* :

```
1 print(len("anticonstitutionnellement"))
2 25
```

- Ligne 1 : la fonction `len`, fournie par Python, effectue le calcul de la longueur.

3. d'où le terme de *littéral*.

Concaténer deux chaînes

L'opérateur + permet de mettre bout à bout deux chaînes :

```
1 print("bon" + "Jour")
```

qui affiche

```
1 bonJour
```

- Ligne 1 : la concaténation est effectuée avec l'opérateur +.
- Ligne 2 : noter que les deux chaînes **bon** et **Jour** sont placées bout à bout, sans espace entre les chaînes.

"bon" + "Jour" est la concaténation de deux chaînes littérales et n'est pas une chaîne littérale.

d) Chaîne vide

Une chaîne peut être vide autrement dit ne contenir aucun caractère. Une chaîne vide peut être représentée par la chaîne littérale "" (deux guillemets qui n'entourent rien). La chaîne vide est de longueur nulle :

```
1 vide = ""  
2  
3 print(len(vide))
```

```
4 0
```

La chaîne vide peut aussi être notée ' ' c'est-à-dire deux apostrophes côte-à-côte, sans espace entre les deux apostrophes :

```
1 vide = ''  
2  
3 # une chaîne vide a une longueur nulle  
4 print(len(vide))
```

```
5 0
```

Usage de la chaîne vide

La chaîne vide a de nombreux usages. Par exemple, une chaîne vide peut être utilisée pour supprimer des caractères à l'intérieur d'une chaîne en remplaçant les caractères par une chaîne vide.

e) Concaténation de chaînes

On peut concaténer des chaînes c'est-à-dire les mettre bout à bout. La concaténation de chaînes est effectuée avec l'opérateur + :

```

1 a = "ali"
2 b = "baba"
3 print(a)
4 print(b)
5 print()
6
7 s = a + b
8 print(s)
9 print()
10
11 print(a)
12 print(b)

```

```

13 ali
14 baba
15
16 alibaba
17
18 ali
19 baba

```

- Ligne XX : `s` est la concaténation des chaînes `a` et `b`
- Lignes XX : la concaténation des chaînes `a` et `b` crée une nouvelle chaîne et préserve les chaînes initiales `a` et `b`.

On peut concaténer davantage que deux chaînes :

```

1 print("abra" + "ca" + "dabra")

```

```

2 abracadabra

```

La concaténation de chaînes crée une nouvelle chaîne, les chaînes concaténées restant intactes.

f) Répétition de chaînes

On peut répéter une chaîne avec l'opérateur de multiplication et un entier positif ou nul :

```

1 s = "Rose"
2 r = s * 10
3 print(r)
4 print(s)

```

```

5 RoseRoseRoseRoseRoseRoseRoseRoseRoseRose
6 Rose

```

- Noter que c'est pas la chaîne initiale `s` qui a été complétée mais une nouvelle chaîne a véritablement été créée.

Ordre des facteurs

Dans le code-source, l'entier peut tout aussi bien multiplier la chaîne par la gauche que par la droite :

```

1 s = "Rose"
2 r = 2 * s * 5
3 print(r)
4 RoseRoseRoseRoseRoseRoseRoseRoseRoseRose

```

Produit par 0

Multiplier par 0 retourne la chaîne vide :

```

1 s = "hello" * 0
2 print(len(s))
3 0

```

Concaténation et répétition

On peut cumuler les opérations de concaténation et de répétition :

```

1 print(3 * "Orange" + 2 * "Kiwi")
2 OrangeOrangeOrangeKiwiKiwi

```

g) Accès aux caractères d'une chaîne

On peut accéder en lecture individuellement aux caractères d'une chaîne à l'aide d'un indice entier :

```

1 p = "Jupiter"
2
3 # le premier caractère de la chaîne p
4 print(p[0])
5
6
7 # le troisième caractère de la chaîne p
8 print(p[2])
9
10
11 # la longueur de la chaîne
12 n = len(p)
13 print(n)
14
15 # Dernier caractère de la chaîne p
16 print(p[n - 1])
17 J
18 p
19 7
20 r

```

Les caractères d'une chaînes sont numérotés de la gauche vers la droite. Ces numéros sont appelés des *indices* ; la numérotation commence à 0 (et non pas à 1) et se termine à n-1 où n désigne le nombre total de caractères.

On accède à chaque caractère de la liste avec l'opérateur `[]` d'indexation.

Le fait de pouvoir accéder à tous les caractères d'une chaîne par des entiers consécutifs se traduit en disant que les chaînes de caractères sont du type *séquence*.

h) Dépassement d'indice dans une chaîne

```
1 z = "orange"
2 c = z[10]
3
4 print(c)

5 Traceback (most recent call last):
6   File "indexError_str.py", line 2, in <module>
7     c = z[10]
8 IndexError: string index out of range
```

- Une erreur d'indice est signalée
- `c` désignerait le caractère à l'indice 10 alors que l'indice maximal dans `c` est 5
- L'opération `z[10]` est en fait interdite, on tente d'accéder en lecture à un caractère qui n'existe pas.
- Le message d'erreur explique que l'indice 10 est en dehors de la plage d'indices possibles.

Tenter d'accéder en simple lecture à un caractère d'une chaîne avec un indice ne correspondant pas à un élément de la chaîne conduit à une erreur, de type `indexError` et qualifiée de *débordement d'indice*.

i) Créer une chaîne à partir de la chaîne vide

Soit à construire une chaîne `z` dont les caractères sont exactement les consonnes d'une chaîne `s` donnée. Par exemple, si `s` est la chaîne

`broccoli`

alors la chaîne cherchée est `brccll`.

Pour cela, il suffit de parcourir la chaîne `s` et d'ajouter à une chaîne initialement vide (disons `z`) le caractère courant de `s` si le caractère est une consonne (et donc s'il n'est pas une voyelle). D'où le code

```
1 s= "broccoli"
2 VOYELLES ="aeiouy"
3 z = ""
4
5 for c in s:
6     if c not in VOYELLES:
7         z += c
8
9 print(s)
10 print(z)

11 broccoli
12 brccll
```

- Lignes 3, 10 et 12 : la chaîne à construire est initialement vide.
- Lignes 5-7 : la chaîne `s` est agrandi au fur et à mesure par ajout d'une consonne.

j) Boucle for : parcours de chaînes

Une chaîne est une séquence et peut être parcourue par une boucle for. Voici un exemple de parcours d'une chaîne avec une boucle for :

```
1 for c in "alibaba":  
2     print(c.upper())
```

```
3 A  
4 L  
5 I  
6 B  
7 A  
8 B  
9 A
```

k) Méthode count

La méthode de chaîne `count` permet de compter le nombre d'occurrences d'une sous-chaîne dans une chaîne :

```
1 s = "jbABCcskpABCknczABCnedz"  
2 p = s.count("ABC")  
3 print(p)
```

```
4 3
```

On notera que la méthode `count` entraîne le parcours caractère par caractère de la chaîne à examiner.

l) Recherche de sous-chaîne avec l'opérateur in

On peut tester la présence d'un sous-chaîne dans une chaîne à l'aide de l'opérateur `in` :

```
1 s = "jbABCcskpABCknczABCnedz"  
2 print("ABC" in s)
```

```
3 True
```

Ce type de test n'est pas adapté si on a besoin de connaître la place éventuelle de la sous-chaîne dans la chaîne.

EXERCICES

Verbes du premier groupe

On donne une liste de verbes à l'infinitif, écrits en minuscule. Écrire un programme qui, affiche les verbes de la liste qui sont du premier groupe, c'est-à-dire se terminant en **er**, comme *chanter*, *manger*, etc. Par exemple, si la liste est formée des verbes

boire, chanter, fuir, voir, avancer, lire

le programme doit afficher juste :

```
1 chanter  
2 avancer
```

Chaînes formées seulement de lettres parmi a ou b

On appellera *babachaine* toute chaîne constituée de caractères uniquement parmi les deux lettres minuscules **a** ou **b**. Par exemple, la chaîne **bbabaaab** ou encore la chaîne **bb** sont des *babachaine*, mais pas **dada** (qui contient le caractère **d**).

- ① Soit une *babachaine* **C**. Écrire un code Python qui renvoie le nombre d'occurrences de la lettre **a** dans la chaîne **C**. Par exemple,
 - si **C** est la chaîne **ababa**, le programme doit renvoyer 3 ;
 - si **C** est la chaîne **bbb**, le programme doit renvoyer 0.
- ② Soit une chaîne *babachaine* **C** de longueur impaire. Écrire un code Python qui renvoie la lettre la plus présente dans la chaîne **C**. Par exemple,
 - si **C** est la chaîne **ababa**, le programme doit renvoyer **a** ;
 - si **C** est la chaîne **bbb**, le programme doit renvoyer **b**.

Palindrome

- ① À partir d'une chaîne de caractères, comme **citoyen**, construire la chaîne inversée, dans l'exemple, cela donne **neyotic**.
- ② Le mot **RADAR** est un *palindrome* : quand on lit ses lettres de la droite vers la gauche, le mot est inchangé. Dire si un mot est oui ou non un palindrome.

Doubler les lettres

À partir d'une chaîne telle que **bali**, on veut construire la chaîne **bbaallii**.

Plus précisément, étant donné une chaîne de caractères, sans accent, écrire une nouvelle chaîne telle que chaque caractère de la chaîne initiale soit doublé.

Suite de Prouhet

La suite de Prouhet est la suivante :

```
1 0
2 01
3 0110
4 01101001
```

Cette suite est une suite de chaînes composées de 0 et de 1 de la manière suivante : chaque chaîne **T** s'obtient à partir de la précédente **U** en adjoignant à **U** la suite **V** obtenue à partir de **U** en échangeant tout 0 en 1 et tout 1 en 0.

Quelle est la 20^e chaîne de la suite de Prouhet.

Créer un mot de passe

Un mot de passe **m** est considéré comme sûr si les conditions suivantes sont réalisées :

- **m** est formé d'au moins huit caractères ;
- **m** contient au moins :
 - une lettre minuscule
 - une lettre majuscule
 - un chiffre
 - un signe de ponctuation parmi le tiret, le point d'exclamation ou d'interrogation, la virgule, le point, le point-virgule, le deux-points, l'astérisque et le tilde.

Créer une variable `mot_de_passe_valide` qui vaut `True` si le mot de passe `m` est sûr et `False` sinon.

Conjuguer un verbe du premier groupe

On donne un verbe du premier groupe, par exemple, *danser*. Ecrire un code qui affiche la conjugaison de ce verbe au présent de l'indicatif. Par exemple, avec *danser*, le code affichera :

```
1 je danse
2 tu danses
3 il/elle danse
4 nous dansons
5 vous dansez
6 ils/elles dansent
```

5 Boucles imbriquées

a) Boucles for imbriquées

Soit à écrire un code Python qui affiche un texte comme le suivant :

```
1 1
2 1 2
3 1 2 3
4 1 2 3 4
5 1 2 3 4 5
6 1 2 3 4 5 6
7 1 2 3 4 5 6 7
8 1 2 3 4 5 6 7 8
9 1 2 3 4 5 6 7 8 9
10 1 2 3 4 5 6 7 8 9 10
11 1 2 3 4 5 6 7 8 9 10 11
12 1 2 3 4 5 6 7 8 9 10 11 12
13 1 2 3 4 5 6 7 8 9 10 11 12 13
14 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Plus précisément, chaque ligne `L` reprend la ligne précédente `K` mais `L` est complétée par l'entier qui suit le dernier entier de la liste affichée dans `K`.

Noter qu'il y a deux types de répétition d'actions :

- répétition d'affichage de ligne
- répétition d'affichage de nombres dans chaque ligne

Un code répondant au problème pourrait être le suivant :

```
1 for lig in range(1,15):
2     for col in range(1,lig + 1):
3         print(col, end = " ")
4     print()
```

On constate qu'une boucle `for` (lignes 2-3) est imbriquée dans une boucle `for` (lignes 1-4). On dit que :

- la boucle ligne 1 est la *boucle externe* ;

— la boucle ligne 2 est la *boucle interne* (ou encore la boucle la plus *profonde*).

La boucle **for** (ligne 1) contrôle la progression par ligne. Un numéro **lig** de ligne étant fixée, l'autre boucle **for** (ligne 2) contrôle la progression de gauche à droite dans la ligne fixée.

Noter à la ligne 3 la présence de l'argument nommé de la fonction **print** :

```
end = " "
```

qui permet

- d'empêcher le passage à la ligne pendant qu'on affiche les éléments de la i^e ligne,
- de séparer deux nombres par un espace.

Noter aussi ligne 4, l'appel à la fonction **print** pour séparer deux lignes consécutives de l'affichage.

D'une façon générale, le corps de toute boucle **for** peut contenir n'importe quel type d'instructions, et en particulier une autre boucle **for**.

b) Effectuer plusieurs tests à l'aide d'une boucle **for**

Pour conforter la validité d'un programme, il est d'usage d'effectuer des tests. Pour tester un morceau de code sur un grand nombre d'entrées, on peut utiliser une boucle **for** qui parcourt une liste de données à tester.

Soit un code qui doit tester si un entier $n \geq 0$ est strictement plus grand que 1000 ou, sinon, multiple de 10 mais pas multiple de 100. Par exemple, 2021 vérifie le test car $2021 > 1000$, 120 aussi (c'est un multiple de 10 mais pas de 100) mais pas 700 car $700 < 1000$ et 700 est multiple de 100.

D'une façon générale, pour vérifier le caractère correct d'un programme, on ne se limite pas à un ou deux tests de ce programme. On accumule les tests en ayant la volonté de mettre en défaut le programme :

```
1 tests = [700, 20, 42, 2020, 2021, 3000, 1000, 0]
2 for i in range(len(tests)):
3     x=tests[i]
4     if x > 1000 or (x % 10 == 0 and x % 100 != 0):
5         print(x, True)
6     else:
7         print(x, False)
```

```
8 700 False
9 20 True
10 42 False
11 2020 True
12 2021 True
13 3000 True
14 1000 False
15 0 False
```

- Ligne 1 : les données à tester sont placées dans une liste
- La totalité du code à tester (lignes 3-7) est placée dans le corps de la boucle **for** parcourant la liste (ligne 2).

- Lignes 5 et 7 : la sortie (lignes 8-15) doit être lisible pour qu'on puisse vérifier rapidement l'exactitude de la réponse donnée par le programme (c'est pour cela que `x` est affiché)
- Ligne 1 : on essaye de mettre en défaut le programme en examinant tous les cas typiques possibles et en étudiant les valeurs particulières situées aux bords comme ici 0, 1000.

EXERCICES TYPE

Grille : remplissage par lignes

Vous allez devoir écrire un code qui affiche un motif dont un exemple est visible ci-dessous :

```

1 42 43 44 45
2 46 47 48 49
3 50 51 52 53
4 54 55 56 57
5 58 59 60 61
6 62 63 64 65
7 66 67 68 69
8 70 71 72 73
9 74 75 76 77
10 78 79 80 81

```

Ce motif consiste en une grille rectangulaire de dimensions $n \times p$ données (n : nombre de lignes, p : nombre de colonnes) de tous les entiers consécutifs à partir d'un entier donné d , le remplissage devant se faire *ligne par ligne*.

Ci-dessus, l'affichage est obtenu pour $n=10$, $p=4$ et $d=42$. Noter que deux colonnes sont séparées par un espace.

Solution

Un motif 2D nécessite typiquement deux boucles imbriquées. Ici, on doit répéter n fois l'écriture d'une ligne et chaque ligne est formée de la répétition des p éléments de la ligne. On a donc un schéma du type :

```

1 for i in range(n):
2     for j in range(p):
3         # A COMPLÉTER

```

D'où, par exemple, le code suivant :

```

1 n=10
2 p=4
3 d=42
4 k=0
5 for i in range(n):
6     for j in range(p):
7         print(d+k, end=' ')
8         k+=1
9     print()

```

Liste d'entiers sans éléments consécutifs

On donne une liste L d'entiers et on cherche à construire un booléen `sansConsecutifs` et qui vaut `True` si L ne contient pas deux entiers consécutifs. Voici en fonction de L différentes valeurs pour la booléen `sansConsecutifs`

```
1 [9, 2, 0, 4, 4, 0, 7] -> False
2 [9, 5, 0, 4, 4, 0, 7] -> True
3 [7, 2, 0, 4, 4, 3, 7] -> True
4 [7] -> False
```

Solution

On rappelle que *consécutif* signifie qu'il y a 1 d'écart, comme entre 2020 et 2021 ou entre 42 et 41.

L'idée est d'essayer tous les cas : on compare le premier élément de L à chaque élément de L et on regarde s'il y aurait 1 d'écart entre les éléments et si oui, on met à jour un drapeau `sansConsecutifs` en le faisant passer à `True` (il est placé à `False` au départ). Puis on recommence avec le deuxième élément de L.

Dans le cas du 2^e exemple ci-dessus, il y aura 1 d'écart avec le 4^e élément de L et donc le drapeau va changer. Et ainsi de suite jusqu'à épuisement de la liste.

Si après toutes ces comparaisons, le drapeau est resté inchangé (à `False` donc), c'est qu'il n'y a jamais deux éléments consécutifs dans la liste et si au contraire, il a un moment basculé à `True`, c'est que la liste contient deux éléments consécutifs.

D'où le code suivant :

```
1 tests=[
2     [9, 2, 0, 4, 4, 0, 7],
3     [9, 5, 0, 4, 4, 0, 7],
4     [7, 2, 0, 4, 4, 3, 7],
5     [7]]
6
7
8 for L in tests:
9     sansConsecutifs=False
10    for a in L:
11        for b in L:
12            if a-b==1:
13                sansConsecutifs=True
14    print(L, '->', sansConsecutifs)
```

Signalons juste que ce code présente plusieurs défauts :

- il compare systématiquement l'écart entre a et b puis entre b et a, ce qui est inutile ;
- les boucles continuent à tourner si le drapeau est passé à `True` ce qui évidemment n'est pas optimal.

EXERCICES

Répéter l'affichage du contenu d'une liste

Soit une liste L d'entiers (entre 0 et 9, pour des raisons d'esthétique de l'affichage). On demande d'afficher n fois cette liste où $n > 0$ est un entier donné. Par exemple, si L est la liste de contenu :

```
1 4 8 0 9 6 3 2 8
```

et si $n = 6$, l'affichage obtenu doit être :

```
1 4 8 0 9 6 3 2 8
2 4 8 0 9 6 3 2 8
3 4 8 0 9 6 3 2 8
4 4 8 0 9 6 3 2 8
5 4 8 0 9 6 3 2 8
6 4 8 0 9 6 3 2 8
```

Deux « cases » successives seront séparées d'une espace.

Damier de nombres

On dit que deux entiers a et b ont même parité si a et b sont tous les deux pairs ou bien si a et b sont tous les deux impairs. Par exemple, 81 et 31 ont même parité, 12 et 82 ont même parité et 81 et 12 ont des parités différentes.

- ① Ecrire un code qui à partir de deux entiers calcule la valeur d'une variable **memeParite** qui vaut **True** si les deux entiers ont même parité, et **False** sinon.
- ② Réaliser un programme qui affiche un damier de forme carrée, de côté de longueur $n > 0$ et rempli alternativement du nombre 4 et du nombre 2. La case en haut à gauche sera toujours 4. Par exemple, pour $n = 7$, le damier aura l'allure suivante :

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

Somme de sommes

Soit la suite (U) des entiers positifs qui ne s'écrivent, en base 10, qu'avec le chiffre 1 :

1, 11, 111, 1111, 11111, ...

On remarquera qu'un tel nombre est une somme de puissances de 10, par exemple pour *mille cent-onze* :

$$1111 = 10^0 + 10^1 + 10^2 + 10^3 = 1 + 10 + 100 + 1000.$$

- ① Soit $n \geq 1$ un entier donné. Calculer, avec une boucle **for**, le n -ième nombre de la suite (U) (on ne se contentera pas d'afficher le nombre mais on le placera dans une variable). Par exemple, si $n = 4$, le programme doit calculer le nombre 1111.
- ② Calculer la somme des N premiers nombres de la suite (U) . Par exemple, si $N = 20$:

$$1 + 11 + 111 + 1111 + 11111 + \dots + 11111111111111111111$$

Pour cela, on placera une partie du code de la question précédente dans le corps d'une boucle **for**.

Somme qui vaut 42

On donne deux listes d'entiers L et M. Créer un booléen **somme42** qui vaut True si la somme de deux nombres, l'un dans L et l'autre dans M, vaut 42. Sinon, le booléen vaudra False. Voici quelques exemples de comportements attendus :

```
1 [17, 22, 5, 5, 33, 8] [34, 8, 20] -> True
2 [6, 22, 5, 5, 33, 8] [35, 8, 25] -> False
```

Afficher les effectifs d'une liste d'entiers

On donne une liste L d'entiers, par exemple

`L = [81, 31, 81, 12, 81, 9, 12, 65]`

On demande d'afficher le nombre de fois que les différents nombres de la liste L apparaissent dans L. L'ordre d'affichage devra respecter l'ordre d'apparition dans la liste L. Avec l'exemple ci-dessus, le programme devra afficher

```
1 81 : 3
2 31 : 1
3 12 : 2
4 9 : 1
5 65 : 1
```

La ligne

```
1 12 : 2
```

de l'affichage signifie juste que 12 apparaît 2 fois dans la liste L.

Multiple de 42 qui soit somme de deux carrés

Un carré⁴ est un nombre qui est de la forme $n \times n = n^2$ où n est un entier positif, par exemple 36 ou 49 sont des carrés mais pas 42.

Certains entiers peuvent s'écrire comme la somme de deux carrés. Par exemple, le nombre $13 = 9 + 4$ est une somme des carrés 9 et 4. En revanche, en essayant les différents cas possibles :

$$12 = 0 + 12 = 1 + 11 = 4 + 8 = 9 + 2$$

on voit que 12 ne peut pas s'écrire comme une somme de deux carrés.

Il se trouve qu'il existe un unique entier entre 1 et 1000 qui est multiple de 42 et peut s'écrire comme somme de deux carrés. On vous demande de trouver ce nombre.

On appliquera la méthode suivante. On observe d'abord que $32^2 = 1024 > 1000$ donc une solution du problème est la somme de carrés de nombres inférieurs à 32. Ensuite, on remarque qu'il suffit d'examiner tous les cas possibles en calculant toutes les sommes de la forme $a^2 + b^2$ où a et b sont des entiers strictement inférieurs à 32.

Supprimer les doublons

Ecrire une fonction `eliminer_doublons(L)` qui partant d'une liste d'entiers retourne une liste composée des valeurs de L, mais dans laquelle les multiples valeurs égales de L (s'il y en a) n'ont été copiées qu'une seule fois. On dira ainsi de la liste retournée qu'elle ne contient pas de doublon. Exemples :

4. Sous entendu, « carré parfait ».


```

1 [42, 81, 42, 65, 12, 81, 31, 42] -> [42, 81, 65, 12, 31]
2 [42] -> [42]
3 [42, 42, 42, 42, 42] -> [42]

```

Motif carré formé des chiffres 4 ou 2

Écrire un programme qui à partir d'un entier $n > 0$ affiche un carré de côté n , dont la bordure est faite avec le nombre 4 et l'intérieur rempli par le nombre 2. Deux chiffres successifs seront séparés par une espace.

Voici un exemple avec un carré de côté $n = 8$:

```

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

```

Triangle de Floyd

- ① Le triangle suivant est un triangle de Floyd à 5 lignes :

```

1 1
2 2 3
3 4 5 6
4 7 8 9 10
5 11 12 13 14 15

```

Le triangle de Floyd à n lignes est obtenu en plaçant sur n lignes successives, dans l'ordre croissant, des entiers consécutifs à partir de 1, la ligne numéro k comportant exactement k entiers. Écrire une fonction `floyd` qui prend en paramètre un nombre de lignes $n > 0$ et affiche le triangle de Floyd à n lignes. Par exemple, l'appel `f(5)` affichera le triangle ci-dessus.

- ② Écrire une fonction `lig_col_floyd(N)` qui prend en paramètre un entier N et renvoie, sous forme de liste, le numéro de ligne et le numéro de colonne dans le triangle de Floyd de la position où se trouve l'entier N . Par exemple, `lig_col_floyd(13)` doit renvoyer la liste `[5, 3]` car 13 se trouve à la 5^e ligne et en 3^e position dans sa ligne. De même, `lig_col_floyd(4)` vaut `[3, 1]`.

Somme nulle

On donne une liste L d'entiers et on demande de construire un booléen `somme0` valant `True` s'il existe des entiers successifs dans L et dont la somme vaut 0. Par exemple, si $L = [-4, 2, -4, 1, 9, -6, -4]$ alors `somme0 = True` puisque $(-4) + 1 + 9 + (-6) = 0$. De même, si $L = [-4, 0, 5, 1]$ alors `somme0 = True` puisque la liste contient un terme nul. En revanche, si $L = [-3, 2, 4]$ alors `somme0 = False`.

Grille : remplissage par colonnes

Vous devez écrire un code qui affiche un motif dont un exemple est visible ci-dessous :

1	42	52	62	72
2	43	53	63	73
3	44	54	64	74
4	45	55	65	75
5	46	56	66	76
6	47	57	67	77
7	48	58	68	78
8	49	59	69	79
9	50	60	70	80
10	51	61	71	81

Ce motif consiste en une grille rectangulaire de dimensions $n \times p$ données (n : nombre de lignes, p : nombre de colonnes) de tous les entiers consécutifs à partir d'un entier donné d , le remplissage devant se faire *colonne par colonne*.

Ci-dessus, l'affichage obtenu pour $n=10$, $p=4$ et $d=42$. Noter que deux colonnes sont séparées par un espace.

Vous pourrez remarquer que dans chaque ligne, les valeurs des nombres se suivent avec un écart valant le nombre p de colonnes, par exemple 42, 52, 62 et 72 dans la première ligne.

6 Boucle while

a) Boucle while

Un problème typique

Étant donné un entier n , on cherche un entier noté m qui soit le premier multiple de 10 supérieur ou égal à n . Si par exemple $n = 42$ alors $m = 50$: en effet, 50 est bien un multiple de 10 et il n'y en pas d'autre entre 42 et 50. Voici d'autres exemples :

n	2038	420	0
m	2040	420	0

L'idée à implémenter

Les multiples de 10 sont les entiers : 0, 10, 20, 30 etc. Ils s'écrivent sous la forme $10k$ où k varie à partir de 0.

Pour trouver m , on parcourt les multiples de 10 depuis 0 et on s'arrête une fois qu'on a atteint ou dépassé l'entier n donné. Autrement dit, on énumère les multiples m de 10 **tant que** $m < n$. Le parcours est montré dans le tableau ci-dessous où on suppose que $n = 42$.

k	0	1	2	3	4	5
$10k < 42 ?$	oui	oui	oui	oui	oui	non

Implémentation en Python

La notion de « dès que », « une fois que » ou de « tant que » est traduite dans le code Python par l'instruction **while**.

Le code suivant répond au problème posé :

```

while_typique.py
1 n = 42
2 k = 0
3
4 while 10 * k < n:
5     k = k + 1
6
7 print(10 * k)
8 50

```

On voit (cf. lignes 7 et 8) que la solution au problème est 50.

Analyse de code

On passe en revue le code de `while_typique.py`.

Vocabulaire de la boucle `while`

- Lignes 4-5 : ces deux lignes forment une seule et même instruction, dite *instruction* `while`.
- Ligne 4 : l'*en-tête* de la boucle `while` est la partie qui commence avec `while` et se termine avant le séparateur `:` (deux-points).
- Ligne 5 : le *corps* de la boucle `while` qui est le bloc indenté sous les deux-points.

Répétition

Une boucle `while` répète une action (ligne 5) tant qu'une *condition*, ici (cf. ligne 4) :

$$10 * k < n$$

reste vraie. Plus précisément,

- si la condition est vraie, l'exécution entre dans le corps de la boucle et à la fin du corps de la boucle, la condition est à nouveau testée (d'où le terme de *boucle*)
- si la condition est fausse, l'exécution du programme va au-delà du corps de la boucle, ici à la ligne 6.

Variable de contrôle

Une boucle `while` fait souvent appel à une *variable de contrôle*, ici `k`, qui évolue pendant la boucle. Typiquement,

- cette variable est **initialisée** avant la boucle, ici ligne 2 ;
- cette variable est **réaffectée**, dans le corps de la boucle `while`, ici par l'instruction ligne 5 ;
- la condition à tester (ici ligne 4) est différente d'un tour de boucle à l'autre car elle dépend de `k` qui, entre-temps, a varié.

Remarques

- Ligne 4 : dans l'immense majorité des cas, le corps d'une boucle `while` est indenté.
- Ligne 1 : il est possible de faire d'autres essais de code en changeant juste la valeur de `n`. Par exemple, si on change `n` en 2038, le programme affichera 2040.

- Il se peut que l'exécution du code n'entre même pas dans le corps de la boucle **while**. Par exemple, si ligne 1, on choisit `n=0` au lieu de `n=42`, le test `10 * k < n` (ligne 4) échoue **immédiatement** et donc le programme continue à la ligne 6 sans même passer par la ligne 5.
- À la différence de la syntaxe des boucles `while` du C et de Java, le booléen évalué avant chaque tour de boucle n'a pas besoin d'être entouré de parenthèses.

Comprendre comment s'exécute une boucle **while**

Modifions le code précédent `while_typique.py` pour mieux comprendre l'exécution de la boucle **while** :

`while_typique_affichage.py`

```

1 n = 42
2 k = 0
3 print("avant while", "-> k=", k)
4 print()
5
6 while 10 * k < n:
7     print("debut while", "k=", k, "-> 10*k=", 10*k)
8     k = k + 1
9     print("fin while", "-> k=", k)
10    print()
11 print("apres while")
12 print(10 * k)

```

```

13 avant while -> k= 0
14
15 debut while k= 0 -> 10*k= 0
16 fin while -> k= 1
17
18 debut while k= 1 -> 10*k= 10
19 fin while -> k= 2
20
21 debut while k= 2 -> 10*k= 20
22 fin while -> k= 3
23
24 debut while k= 3 -> 10*k= 30
25 fin while -> k= 4
26
27 debut while k= 4 -> 10*k= 40
28 fin while -> k= 5
29
30 apres while
31 50

```

En plus du code initial, `while_typique_affichage.py` contient des instructions d'affichage (lignes 3, 7, 9 et 11) et des instructions de sauts de ligne (lignes 4 et 10) dans la sortie pour observer l'évolution de `k` et de `10 * k` avant, pendant et après la boucle **while**.

L'exécution du programme est la suivante :

- Lignes 2 et 7 : la valeur de `k` avant le commencement de la boucle **while**

- Ligne 6 : `k` vaut 0. Le test de la boucle `while` est effectué : a-t-on $0 < 42$? La réponse est *oui* donc, l'exécution du code continue dans le corps de la boucle `while`
- Les affichages sont effectués : `k` est changé de 0 à 1 (lignes 8 et 16).
- Ligne 6 : le test de la boucle `while` est à nouveau effectué : a-t-on $10 < 42$? La réponse est *oui* et l'exécution entre à nouveau dans le corps de la boucle. Cette opération se répète jusqu'à ce que `k = 5` (ligne 8 et ligne 28).
- Ligne 6 : le test de la boucle `while` est effectué : a-t-on $50 < 42$? Cette fois, la réponse est *non* donc l'exécution quitte la boucle et continue lignes 11 et 12, cf. lignes 30 et 31.
- Ligne 12 : le résultat affiché (cf. ligne 31) est bien le résultat demandé. Comme le test $10 * k < n$ a échoué pour la première fois, c'est qu'on a $10 * k \geq n$ avec `k` minimal et c'est bien ce que l'on cherchait.

b) Boucle for vs boucle while

Python dispose de deux types de boucle : la boucle `for` et la boucle `while`. Elles ont chacune des usages spécifiques.

Usage canonique de la boucle for

La boucle `for` s'emploie si :

- une action doit être effectuée un nombre prédéterminé de fois, et, surtout, connu avant l'exécution de la boucle. Par exemple, s'il faut afficher 42 nombres vérifiant une certaine propriété, on utilise une boucle du type `for i in range(42)`.
- on parcourt une liste entre deux indices connus à l'avance, ce qui peut être la totalité de la liste ou, par exemple, de la 5^e position à l'avant-dernière.

Par exemple, étant donné une liste `L` de nombres, si on cherche à déterminer combien `L` contient d'entiers positifs, on utilisera une boucle d'en-tête du type `for x in L` ou encore `for i in range(len(L))`.

Usage canonique de la boucle while

La boucle `while` s'emploie si une action doit être répétée tant qu'une certaine condition reste vraie et sans qu'on sache à l'avance combien de fois la boucle va être répétée. Par exemple,

- on a une liste d'entiers et on cherche le **premier** entier de la liste qui soit pair ;
- on cherche la plus petite puissance de 10 qui dépasse un entier donné.

Usage inapproprié d'une boucle while

Le programme ci-dessous utilise une boucle `while` pour afficher les 5 premiers multiples de 10 :

```

1 i = 1
2 while i <= 5:
3     print(10 * i)
4     i = i + 1
5
6 print("Fin")

```

```

7 10
8 20
9 30
10 40
11 50
12 Fin

```

Le code `while_inapproprié.py` est un usage non recommandé de la boucle `while`. En effet, le code contient des instructions inutiles par rapport à une boucle `for` :

- Ligne 1 : initialisation inutile, c'est automatique avec une boucle `for`.
- Ligne 2 : test de condition, automatique avec la boucle `for`.
- Ligne 4 : incrémentation inutile, c'est automatique avec une boucle `for`.

Comparer avec un code équivalent écrit avec une boucle `for` :

```

1 for i in range(5):
2     print(10 * i)
3
4 print("Fin")

```

c) Boucle `while` et parcours de liste

Usuellement, le parcours d'une liste est associé à une boucle `for`. Toutefois, dans certaines circonstances, il est plus approprié de parcourir une liste avec une boucle `while`.

Soit à définir un code Python qui, partant d'une liste `L` d'entiers détermine si `L` ne contient que des entiers positifs. Plus précisément, le code doit définir un booléen nommé `tousPositifs` qui vaut `True` si `L` ne contient que des entiers positifs et `False` sinon.

L'idée est de parcourir la liste `L` avec une boucle `while` à l'aide d'un indice `i` **valide** et tant que le terme courant `L[i]` est positif ou nul :

```

1 L= [31, 82, -42, 32, -81]
2
3 i=0
4 n = len(L)
5
6 while i<n and L[i]>=0:
7     i = i + 1
8
9 tousPositifs = (i==n)
10
11 print(L, tousPositifs)
12 [31, 82, -42, 32, -81] False

```

- Ligne 6 : la condition doit D'ABORD gérer la sortie de liste d'où la condition `i < n` qui évite le débordement d'indice.
- Ligne 6 : la condition doit aussi filtrer suivant le critère, ici le fait que chaque terme soit positif.
- Ligne 9 : Il s'agit de définir le booléen.

- Si un élément négatif est apparu dans la liste, soit i son indice. Alors comme le booléen $i < n$ and $L[i] \geq 0$ vaut `False` (puisque $L[i] < 0$), la boucle `while` s'arrête et l'indice i vérifie $i < n$.
- En revanche, si aucun élément de la liste n'est strictement négatif, le corps de la boucle `while` sera exécuté tant que la condition $i < n$ sera vraie. Lorsque $i = n-1$, le corps de la boucle est exécuté et i devient $i+1$ donc $i=n$ et la boucle se termine.

Donc, ce qui différencie les deux cas, c'est la valeur de i en sortie de boucle : dans le second cas, i vaut n et pas dans le premier cas, ce qui explique la définition ligne 9 de `tousPositifs`.

Noter que le code suivant devrait être évité :

```

1 L= [31, 82, -42, 32, -81]
2
3 i=0
4
5 while i< len(L) and L[i]>=0:
6     i = i + 1
7
8 tousPositifs = (i==len(L))
9
10 print(L, tousPositifs)
```

- Ligne 5 : à chaque tour de boucle, l'appel `len(L)` est recalculé alors que `len(L)` est un nombre qui ne varie pas.

d) Boucle `for` vs boucle `while` : tableau récapitulatif

Utiliser une boucle `while` par rapport à une boucle `for` nécessite plus de soin comme le résume le tableau ci-dessous :

	Boucle <code>while</code>	Boucle <code>for</code>
Initialisation indice i	manuelle	automatique (en-tête de la boucle)
Incrémentatation de l'indice i	manuelle	automatique
Sortie de boucle	manuelle (avec drapeau dans l'en-tête)	nécessite des instructions de saut (<code>break</code> , <code>return</code>)
Sortie de liste de longueur n	manuelle ($i < n$)	automatique

e) Boucle `for` : `break`

Il est parfois souhaitable de pouvoir interrompre l'exécution du corps d'une boucle `for` lorsqu'une certaine condition est réalisée. Cette interruption se pratique avec une instruction `break`.

Soit à définir un code Python qui à partir d'une liste L d'entiers détermine si L ne contient que des entiers positifs. Plus précisément, le code doit définir un booléen nommé `tousPositifs` qui vaut `True` si L contient ne contient que des entiers positifs et `False` sinon.

L'algorithme est simple : il consiste à déclarer un drapeau `tousPositifs`, à parcourir la liste et changer le drapeau en `False` puis interrompre le parcours de la liste :

```
1 L= [31, 82, -42, 32, -81]
2
3 tousPositifs = True
4 for i in range(len(L)):
5     if L[i] < 0:
6         tousPositifs = False
7         break
8 print(L, tousPositifs)
9
10 print("=====")
11
12 L= [31, 82, 421, 32, 81]
13
14 tousPositifs = True
15 for i in range(len(L)):
16     if L[i] < 0:
17         tousPositifs = False
18         break
19 print(L, tousPositifs)
20 [31, 82, -42, 32, -81] False
21 =====
22 [31, 82, 421, 32, 81] True
```

- Le même code (lignes 1-10) est testé avec deux listes différentes (lignes 1 et 12).
- Dans chaque cas, on affiche la liste `L` et le booléen `tousPositifs`
- Le parcours de `L` est initié
- Si un terme de la liste est négatif, le drapeau est changé et la liste n'est plus parcourue à cause de l'instruction `break`.

Lorsque l'interpréteur Python rencontre une instruction `break` dans le corps d'une boucle `for`, le code qui suit l'instruction `break` est ignoré et la suite du code à exécuter commence immédiatement après la fin de l'instruction `for` correspondante.

f) Boucles et affectations combinées

Une boucle (`for`, `while`) est fréquemment l'occasion d'utiliser des affectations combinées.

Soit à coder la somme de tous les entiers entre 1 et `n` :

```
1 n=10
2 s=0
3 for k in range(1,n+1):
4     s += k
5 print(s)
6 55
```

- Chaque étape de la boucle modifie la variable `s` en lui ajoutant la valeur courante
- Assez naturellement, on écrira `s += k` au lieu de `s = s + k`.

Le corps d'une boucle contient souvent des réaffectations de variables qui peuvent être transformées en affectation combinées.

EXERCICES TYPE

Premier entier pair

Dans cette question, l'usage de la boucle **for** est inapproprié.

Ecrire un code qui à partir d'une liste *L* d'entiers :

- affiche le premier entier pair rencontré dans la liste *L* si *L* contient au moins un entier pair ,
- affiche le nombre -1 s'il n'existe aucun entier pair dans la liste.

Voici quelques exemples de comportement de la fonction :

```
1 [1, 3, 42, 51, 32, 9] -> 42
2 [1, 3, 42, 51, 33] -> 42
3 [1, 3, 51, 33] -> -1
4 [1, 3, 51, 33, 42] -> 42
5 [42, 82] -> 42
6 [42] -> 42
7 [81] -> -1
8 [] -> -1
```

Solution

Il suffit d'appliquer la technique vue dans le paragraphe *Boucle while et parcours de liste*.

```
1 L=[1, 3, 42, 51, 32, 9]
2
3 i=0
4 n=len(L)
5
6 while i<n and L[i]%2 == 1:
7     i = i + 1
8
9 if (i==n):
10     print(L, "->", -1)
11 else:
12     print(L, "->", L[i])
13 [1, 3, 42, 51, 32, 9] -> 42
```

- Lignes 6-7 : la boucle **while** parcourt la liste *L* jusqu'à ce que la liste soit épuisée ou bien qu'un élément pair soit rencontrée.
- Lignes 9-10 : en sortie de boucle, ou bien l'indice *i* est hors de la liste et dans ce cas la liste ne contient aucun élément pair (si un élément pair est présent uniquement en dernière position alors, en sortie de boucle, *i* vaut *n*-1).

Pour traiter le cas de plusieurs listes, au lieu de recopier du code, on place les liste dans une liste **tests** et on parcourt la liste **tests** pour tester chacune des lignes :

```

1 tests = [
2 [1, 3, 42, 51, 32, 9],
3 [1, 3, 42, 51, 33],
4 [1, 3, 51, 33],
5 [1, 3, 51, 33, 42],
6 [42, 82],
7 [42],
8 [81],
9 []]
10
11 for L in tests:
12     i=0
13     n=len(L)
14
15     while i<n and L[i]%2 == 1:
16         i = i + 1
17
18     if (i==n):
19         print(L, "->", -1)
20     else:
21         print(L, "->", L[i])
22 [1, 3, 42, 51, 32, 9] -> 42
23 [1, 3, 42, 51, 33] -> 42
24 [1, 3, 51, 33] -> -1
25 [1, 3, 51, 33, 42] -> 42
26 [42, 82] -> 42
27 [42] -> 42
28 [81] -> -1
29 [] -> -1

```

La petite exponentielle contre le grand produit

Trouver le plus grand entier $n > 0$ tel que $1.0001^n < 10000n$ (on trouvera $n = 214893$).

Solution

Posons $a = 1.0001$. Comme a est très proche de 1, les puissances de a s'éloignent de 1 mais très lentement. Par exemple :

```

1 a=1.0001
2 print(a**10000)
3 2.7181459268249255

```

Toutefois, comme a est plus grand que 1, a^n peut être rendu aussi grand que l'on veut en choisissant n assez grand.

L'idée est très simple : on compare a^n et $10000 \times n$ en faisant varier n suivant les entiers consécutifs jusqu'à ce que le premier dépasse le second. D'où le code :

```

1 k=1
2
3 while 1.0001**k < 10000*k:
4     k=k+1
5
6 print("n =", k-1)
7 n = 214893

```

- Ligne 1 : l'énoncé demande de trouver un entier $k > 0$.
- Ligne 6 : en sortie de boucle **while** l'entier k trouvé est le premier qui vérifie $1.0001^k \geq 10000 \times k$. C'est l'entier k précédent qui est tel que l'en-tête de la boucle **while** est vrai. C'est donc lui la solution du problème.

On peut bien sûr vérifier que le résultat $n = 214893$ est conforme :

```

1 k = 214893
2 print(k, "->", 1.0001**k < 10000*k)
3
4 k = 214894
5 print(k, "->", 1.0001**k < 10000*k)
6 214893 -> True
7 214894 -> False

```

Liste constante (avec **break**)

Cet exercice doit utiliser une instruction **break** placée dans une boucle **for**.

On donne une liste L et on demande de créer un booléen **tousEgaux** valant **True** si tous les éléments de la liste L sont égaux et **False** sinon. Par exemple si $L = [42, 42, 42]$ alors **tousEgaux** vaudra **True** et si $L = [42, 421, 42, 42]$ alors **tousEgaux** vaudra **False**.

Solution

```

1 for L in [ [42, 42, 42], [42, 421, 42, 42] ]:
2     tousEgaux = True
3     for i in range(1, len(L)):
4         if L[i] != L[0]:
5             tousEgaux = False
6             break
7     print(L, "->", tousEgaux)

```

EXERCICES

Que des 81 puis que des 12

Soit une liste L formée d'abord d'un certain nombre de fois de l'entier 81 puis d'un certain nombre de fois de l'entier 12. Par exemple, $L=[81, 81, 81, 12, 12, 12, 12]$ ou encore $L=[81, 12]$ ou même $L=[12, 12]$ ainsi que $L=[81, 81, 81]$.

Écrire, à l'aide d'une boucle **while**, une variable `indicePremier_12` qui définit le premier indice du terme de L qui vaut 12. Par exemple, si $L = [81, 81, 81, 12, 12, 12, 12]$ alors

`indicePremier_12 = 3`. Si 12 n'apparaît pas dans L, la variable `indicePremier_12` vaudra `n` où `n` est la longueur de la liste L.

Exemples de comportement :

```
1 [81, 81, 81, 12, 12, 12, 12] -> 3
2 [81, 12] -> 1
3 [81, 81] -> 2
4 [12, 12] -> 0
```

Listes « opposées » (boucle `while`)

Ecrire un code qui partant deux listes d'entiers L et M crée un booléen **sontOpposees** valant **True** si les deux listes sont « opposées » et **False** sinon. Deux listes sont considérées comme « opposées » si elles ont le même nombre d'éléments et si, à des indices identiques, elles possèdent des éléments opposés (comme -81 et 81). Voici quelques exemples de comportements attendus :

```
1 [81, -12, 0, -81, -31] [-81, 12, 0, 81, 31] -> True
2                                     [-81] [81] -> True
3                                     [0, 0] [0, 0] -> True
4                                     [ ] [ ] -> True
5                                     [81, -12] [-81, -12] -> False
6                                     [-81, 12, 0] [81, -12] -> False
```

Vous ne devez pas utiliser de boucle `for` mais une boucle `while`

Plus petit entier non nul (boucle `while`)

On donne une liste d'entiers qui ne sont pas tous nuls, par exemple

— `>L = [5, 8, 0, 9, 12, 0, 6, 4]` ou encore `L = [0, 0, 0, 4, 0, 3, 0]`.

Ecrire une fonction `plusPetitNonNul(L)` qui renvoie le plus petit entier de la liste qui soit non nul. Voici quelques exemples de comportement de la fonction :

```
1 [5, 8, 0, 9, 1, 0, 6, 4] -> 1
2 [0, 0, 0, 4, 0, 3, 0] -> 3
3 [5, 8, 1, 6, 4] -> 1
4 [-5, -8] -> -8
5 [0, -5, -8] -> -8
6 [-3, 0, -5, -8] -> -8
7 [0, 5, -8] -> -8
8 [42] -> 42
9 [-42] -> -42
```

On pourra appliquer la méthode suivante :

- chercher à l'aide d'une boucle `while` le plus petit indice `i` de L tel que `L[i]` soit non nul
- chercher le plus petit élément de la liste L à partir l'indice `i`

Pairs d'abord, impairs ensuite

On donne une liste L d'entiers et on demande d'écrire une fonction `f` qui renvoie **True** si dans la liste L apparaissent D'ABORD les entiers pairs de la liste et ENSUITE les entiers impairs de la liste.

Voici quelques exemples de comportements de `f` :

<i>L</i>	Pairs puis impairs ?	Commentaire
[12, 82, 81, 9, 31]	True	D'abord 12, 82 (pairs) puis les impairs
[81, 9, 31]	True	Que des impairs
[12, 82]	True	Que des pairs
[12, 82, 81, 9, 46, 31]	False	9 (impair) est suivi d'un pair (46)

Liste d'entiers en miroir

On donne une liste d'entiers, par exemple $L = [4, 2, 2, 4]$ et on demande de dire si cette liste est en miroir, autrement dit si

- le premier et le dernier élément de L sont égaux,
- le deuxième et l'avant-dernier élément de L sont égaux,
- et ainsi de suite jusqu'à épuisement de la liste.

Dans le cas de la liste $L = [4, 2, 2, 4]$, cette liste est en miroir. Dans le cas de la liste $L = [4, 2, 1]$, la liste n'est pas en miroir.

Pas d'impair (avec **break**)

Soit une liste L formée d'entiers. Écrire, à l'aide d'une boucle **for** et d'une instruction **break**, un booléen `queDesPairs` qui dit si, oui ou non, L ne contient que des entiers pairs.

Comportement attendu :

```

1 [82, 31, 82] -> False
2 [82, 12, 46] -> True
3 [82] -> True
4 [81] -> False

```

Suite croissante d'entiers consécutifs (avec **while**)

L'exercice doit être codé en utilisant une boucle **while**.

Écrire un code qui à partir d'une liste L d'entiers définit une variable booléenne nommée **consecutifs** qui vaut **True** si la liste est constituée d'entiers CONSÉCUTIFS croissants et **False** sinon. Ci-dessous, voici quelques exemples de comportements attendus

```

1 [81, 82, 83] -> True
2 [82, 81, 83] -> False
3 [2013, 2038, 3000] -> False
4 [81] -> True

```

Suite croissante d'entiers consécutifs (avec **break**)

L'exercice doit être codé en utilisant une boucle **for** et une instruction **break**.

Écrire un code qui à partir d'une liste L d'entiers définit une variable booléenne nommée **consecutifs** qui vaut **True** si la liste est constituée d'entiers CONSÉCUTIFS croissants et **False** sinon. Ci-dessous, voici quelques exemples de comportements attendus

```

1 [81, 82, 83] -> True
2 [82, 81, 83] -> False
3 [2013, 2038, 3000] -> False
4 [81] -> True

```

Calculer le nombre de chiffres d'un entier

On donne un entier $n \geq 0$ et on cherche le nombre de chiffres `nchiffres` de n . Par exemple, si $n = 2020$ alors `nchiffres` = 4 ou encore si $n = 42$ alors `nchiffres` = 2.

L'idée pour calculer `nchiffres` est de compter le nombre de divisions successives de n par 10 jusqu'à ce que le quotient (entier) soit nul. Par exemple

- le quotient entier de 2020 par 10 est 202,
- le quotient entier de 202 par 10 est 20,
- le quotient entier de 20 par 10 est 2,
- le quotient entier de 2 par 10 est 0.

C'est parce 2020 a justement 4 chiffres qu'on a effectué 4 divisions successives par 10 avant d'obtenir un quotient nul.

Ecrire un code Python utilisant une boucle `while` et qui détermine `nchiffres` connaissant n .

Voici quelques exemples de comportements

```
1 42 -> 2
2 2020 -> 4
3 10 -> 2
4 7 -> 1
5 0 -> 1
6 741520036365253625145211741523636854198541 -> 42
```

7 Compléments

a) Concaténation de listes

On peut concaténer des listes c'est-à-dire les mettre bout à bout. La concaténation de listes est effectuée avec l'opérateur `+` :

```
1 a = [65, 31, 9]
2 b = [32, 81, 82, 46, 12]
3 print(a)
4 print(b)
5 print()
6
7 s = a + b
8 print(s)
9 print()
10
11 print(a)
12 print(b)
13 [65, 31, 9]
14 [32, 81, 82, 46, 12]
15
16 [65, 31, 9, 32, 81, 82, 46, 12]
17
18 [65, 31, 9]
19 [32, 81, 82, 46, 12]
```

- Ligne 7 : on concatène deux listes en la liste `s`.
- Lignes 11 et 12 : les listes initiales sont inchangées, elles n'ont pas été prolongées ; autrement dit, `s` est une nouvelle liste.

On peut concaténer davantage que deux listes :

```
1 print([65, 31] + [9, 32, 81] + [82, 46, 12])
2 [65, 31, 9, 32, 81, 82, 46, 12]
```

b) Boucle for : parcours de chaînes

Une chaîne est une séquence et peut être parcourue par une boucle `for`. Voici un exemple de parcours d'une chaîne avec une boucle `for` :

```
1 for c in "alibaba":
2     print(c.upper())
3 A
4 L
5 I
6 B
7 A
8 B
9 A
```

c) Boucle infinie

La notion de boucle infinie

ATTENTION ! Ne pas exécuter le programme ci-dessous dans IPython Notebook, il sera impossible de l'interrompre et le programme risque de planter au moins le navigateur exécutant le code.

Le code ci-dessous donne une partie de l'idée de ce que pourrait être une boucle infinie. Un message va s'afficher sans interruption dans une console. Pour cesser l'affichage, l'utilisateur doit appuyer sur les touches CTRL+C (ou CTRL+Z) :

```
1 while True:
2     print("toujours !")
```

```
3 toujours !
4 toujours !
5 toujours !
6 toujours !
7 toujours !
8 toujours !
9 toujours !
10 toujours !
11 toujours !
12 toujours !
13 toujours !
14 toujours !
15 toujours !
16 toujours !
17 toujours !
18 toujours !
19 toujours !
20 toujours !
21 toujours !
22 toujours !
23 toujours !
24 toujours !
25 toujours !
26 toujours !
27 toujours !
28 toujours !
29 toujours !
30 toujours !
31 toujours !
32 toujours !
33 toujours !
34 toujours !
35 toujours !
36 toujours !
37 toujours !
38 toujours !
39 toujours !
40 toujours !
41 toujours !
42 toujours !
43 toujours !
44 toujours !
45 toujours !
46 toujours !
47 toujours !^C
48 Traceback (most recent call last):
49   File "_py", line 2, in <module>
50     print("toujours !")
51 KeyboardInterrupt
```


Une *boucle infinie* est une boucle dont l'en-tête est `while True`. Comme la condition pour rester dans la boucle est toujours True, la boucle n'a pas de raison de terminer, en théorie tout du moins. Ici, exceptionnellement, le programme est interrompu par une intervention manuelle l'utilisateur (CTRL+C).

Exemple d'utilisation

Un utilisateur est invité par le programme à entrer, un par un, des entiers positifs ou négatifs. Si l'utilisateur entre le nombre 0 ou si la somme des nombres qu'il a entrés devient négative, le programme s'arrête après avoir affiché la somme des entiers entrés par l'utilisateur.

Voici deux exemples de session de ce programme :

```
1 Entrer un entier positif ou négatif : 10
2 Entrer un entier positif ou négatif : 20
3 Entrer un entier positif ou négatif : -25
4 Entrer un entier positif ou négatif : 0
5 5
6 Entrer un entier positif ou négatif : 10
7 Entrer un entier positif ou négatif : -20
8 -10
```

On peut résoudre ce problème en utilisant une boucle infinie :

```
1 s=0
2
3 while True:
4     x = int(input("Entrer un entier positif ou négatif : "))
5     s = s + x
6     if x==0 or s <0:
7         print(s)
8         break
9 Entrer un entier positif ou négatif : 10
10 Entrer un entier positif ou négatif : 20
11 Entrer un entier positif ou négatif : -25
12 Entrer un entier positif ou négatif : 0
13 5
```

— Lorsque l'entrée vaut 0 (cf. ligne 3) ou la somme devient nulle (cf. ligne 3 encore), la somme est affichée (lignes 7 et 13) et la boucle est interrompue (cf. ligne 8) et le programme se termine puisqu'aucune instruction ne suit la boucle while.

Alternative

Il est parfois possible de remplacer une boucle infinie en utilisant un simple drapeau. Dans le cas du code précédent, on peut obtenir :

```

1 s=0
2 enCours = True
3
4 while enCours:
5     x = int(input("Entrer un entier positif ou négatif : "))
6     s = s + x
7     if x==0 or s <0:
8         print(s)
9         enCours = False

```

- Création d'un drapeau `enCours`
- Le changement de la valeur du drapeau permet de quitter la boucle.

TAG non IPython ;

d) Utilisation d'une boucle infinie

En Python, une boucle infinie s'écrit avec une boucle `while` et sous la forme courante suivante `while True:` ou parfois `while 1`.

On utilise une boucle infinie dans les conditions suivantes :

- quand on répète une action qui doit se terminer seulement si une condition « compliquée » est vérifiée ;
- quand on crée un générateur
- quand on crée la boucle d'un jeu dans une interface graphique comme Pygame

Une boucle infinie dont la sortie est mal gérée place le programme dans une véritable boucle infinie.

A la différence d'une boucle `while` classique, une boucle infinie évite

- de définir avant la boucle `while` une ou plusieurs variables afin d'exprimer la condition de sortie de la boucle
- de gérer l'évolution de ces variables pendant la boucle.

Pour une boucle infinie, les variables utiles pour déterminer la sortie de la boucle sont souvent définies à l'intérieur de la boucle.

Chapitre IV

Fonctions

1 Le concept de fonction

a) Les fonctions en Python

La notion de fonction

Considérons la transformation T suivante : étant donné un nombre x , la transformation T envoie x sur $100x$. On écrirait en mathématiques :

$$T(x) = 100x.$$

Une fonction au sens de Python est souvent assez proche de la notion mathématique de fonction :

- elle reçoit des données,
- elle traite les données,
- elle renvoie une valeur.

En résumé, retenir cette équation :

$$\text{fonction} = \text{données} + \text{exécution} + \text{retour}$$

Les fonctions en Python

Les fonctions en Python apparaissent sous deux formes, décrites ci-dessous. La première forme a déjà été abordée, pas la deuxième.

Les fonctions built-in

Les fonctions par défaut¹ peuvent être utilisées telles que Python les propose, par exemple la fonction `max` qui donne le maximum d'une séquence de nombres :

```
1 y = max(81, 12)
2 print(y)
3 81
```

- Ligne 1 : la fonction `max` a juste été appelée sur les données 81 et 12 et elle a renvoyé une valeur (le plus grand des deux nombres, placé dans la variable `y`)

1. Ou encore les fonctions *natives* ou *intégrées*, en anglais, *built-in functions*.

Les fonctions « custom »

Les fonctions personnalisées (*custom*), construites par le programmeur lui-même, pour les besoins spécifiques de son programme :

```
1 def f(x):  
2     return 100*x  
3  
4 y = f(42)  
5 print(y)  
6 4200
```

- Lignes 1-2 : ici, **f** désigne une fonction. Le programmeur a *défini* une fonction **f**, très très simple, en écrivant (ligne 2) le code de ce que **f** va faire (**f** multiplie par 100 la valeur **x** qu'elle reçoit et « renvoie » le résultat). C'est le programmeur qui a écrit le code exécutable de la fonction.
- Ligne 4 : le programmeur *appelle* (ie exécute) la fonction **f** qu'il a créée avec la donnée 42. La fonction **f** renvoie le résultat qu'elle a calculé.

Différences

La différence essentielle entre les deux types de fonctions :

- une fonction built-in est immédiatement disponible sans que le programmeur ait eu à la définir ou la programmer,
- une fonction custom doit être imaginée, nommée par le programmeur et le code exécutable de la fonction est écrit par le programmeur.

Type de retour d'une fonction

Une fonction peut retourner une valeur de n'importe quel type, par exemple :

- un entier,
- un flottant,
- une liste,
- une chaîne,
- une autre fonction,
- des objets complexes comme une instance de classe ou des combinaisons d'objets complexes

Une fonction peut aussi ne rien renvoyer.

b) Une fonction typique

Examinons un exemple typique de création de fonction, que l'on va appeler **f** et utilisons cet exemple pour illustrer les mécanismes de *définition* et d'*appel* de fonction qui sont les deux notions-clé à bien comprendre.

On donne deux notes sur 20, disons **a** et **b**, à la fonction **f** et **f** nous renvoie **True** si la moyenne de ces deux notes est supérieure à 10 et **False** sinon.

Pour cela, **f** doit calculer la moyenne

$$m = \frac{a + b}{2}$$

et d'examiner si, oui ou non, l'assertion $m \geq 10$ est vraie ou pas.

Voici un code possible en Python d'une telle fonction :

fonction_typique.py

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
8 True
```

- Lignes 1-4 : le code de la fonction **f**. Ce code est ce qu'on appelle en fait la *définition* de la fonction **f**.
- Ligne 6 : le programme demande à la fonction **f** de calculer s'il est vrai ou pas que les notes 14 et 8 donnent une moyenne supérieure à 10 ou pas. Ce que renvoie la fonction **f** (soit **True**, soit **False** ici) est placé dans la variable **resultat**.
- Ligne 7 : on affiche le booléen **resultat**.
- Ligne 4 : la fonction **f** renvoie un résultat, ici un booléen, à l'aide du mot-clé **return** qui signifie « retourner » ou, mieux, « renvoyer ».

Sémantique d'un appel de fonction

On reprend le code de `fonction_typique.py` ci-dessus :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
8 True
```

Que se passe-t-il lors de l'appel `f(14, 8)` à la ligne 6 ? Réponse : les **paramètres** `a` et `b` (ligne 1) sont remplacés, dans cet ordre, par les **arguments** 14 et 8 dans la définition de la fonction **f** et le code Python de la définition de **f** est exécuté avec les valeurs remplacées.

Autrement dit, une fois l'appel ligne 6 lancé, c'est **comme si** le code suivant était exécuté :

equiv_fonction_typique.py

```
1 a = 14
2 b = 8
3
4 m = (a + b)/2
5 ok = (m >= 10)
6
7 resultat = ok
8
9
10 print(resultat)
```

11 True

- Lignes 1-7 : l'exécution de ce code est équivalente à l'exécution de la fonction `f` lors de l'appel `f(14, 8)`.

Vocabulaire associé à la notion de fonction

On reprend le code de `fonction_typique.py` ci-dessus :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
```

- Ligne 1 : la première ligne de la définition de la fonction est ce qu'on appelle l'*en-tête* de la fonction. Cet en-tête commence toujours par le mot-clé `def`.
- Ligne 1 : l'en-tête se termine par le symbole « deux-points ».
- Lignes 2-4 : la partie qui suit l'en-tête est ce qu'on appelle le *corps* de la fonction `f`. Ce corps est un bloc indenté.
- Ligne 6 : l'expression `f(14, 8)` est ce qu'on appelle un *appel* de la fonction `f`. Cette expression est placée hors de la définition de fonction et est dite *expression appelante*.

Les valeurs que l'on passe à `f` lors de cet appel (ici 14 et 8) s'appellent les *arguments* de l'appel.

- Ligne 1 : dans la définition de la fonction, ici les variables `a` et `b`, s'appellent les *paramètres* de la fonction `f`.
- Ligne 4 : noter que l'objet qui suit le mot-clé `return` n'est pas entouré de parenthèses. Il est possible d'en placer mais ce n'est pas utile.

Les *paramètres* n'existent que dans la définition de la fonction. Les paramètres sont juste des noms qui servent à désigner les valeurs que la fonction va recevoir plus tard, lorsque la fonction sera appelée. Les paramètres ont donc un rôle purement *formel*.

Les arguments ne sont pas des variables, ce sont des valeurs, éventuellement des valeurs de variables. On dit parfois que les arguments sont des paramètres *réels* (autrement dit, non formels).

Le mécanisme d'appel est ce qu'on désigne parfois sous le terme de *passage des arguments* ou encore *transmission des arguments*.

Ordre d'exécution du code

On reprend le code de `fonction_typique.py` ci-dessus :

```

1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)

```

- Ligne 1 : l'interpréteur Python commence à cette ligne la lecture du code. En voyant l'entête de la fonction, il prend en compte la déclaration de la fonction `f`. Autrement dit à partir de ce moment-là, le nom `f` est reconnu comme celui de la fonction qui est définie à partir de la ligne 1. L'exécution du code continue ligne 6 en ignorant le corps de la fonction.
- Lignes 2-4 : ces lignes ne sont pas exécutées car il n'y a pas eu, pour l'instant, d'appel à la fonction `f`.
- Ligne 6 : l'ordre par défaut d'exécution est modifié puisque l'exécution retourne en arrière dans le code source pour exécuter la fonction `f` des lignes 1 à 4.
- Ligne 7 : une fois l'exécution terminée, l'exécution du code passe de la ligne 4 à la ligne 7.

2 Renvoi d'une fonction

a) return et fin d'exécution

Un point essentiel à comprendre concernant le `return` d'une fonction peut être résumé ainsi :

`return` = GAME OVER

autrement dit, une fois l'instruction `return` exécutée, l'appel de la fonction est définitivement interrompu, sans aucune exception.

Illustrons avec le code suivant :

```

1 def f(x,y,z):
2     v= x**2 + y**2 - z**2
3     return v
4
5 t=f(4,3,5)
6 print(t)
7 0

```

Le code de `f` s'exécute ligne par ligne. Lorsque l'exécution arrive à la ligne 3 :

- l'exécution de la fonction `f` s'interrompt,
- la valeur `v` obtenue est « renvoyée » et affectée à `t`.

Tout code figurant dans la définition de `f` après l'instruction `return` (ligne 3) serait ignoré lors de l'exécution, par exemple

```

1 def f(x,y,z):
2     v= x**2 + y**2 - z**2
3     return v
4     v= 42
5
6 t=f(4,3,5)
7 print(t)
8 0

```

— Ligne 4 : cette ligne est comme ignorée et n'est jamais exécutée par `f`.

b) Fonction renvoyant plusieurs valeurs

A proprement parler, une fonction ne peut renvoyer qu'une seule valeur. Cependant, comme une fonction peut renvoyer un conteneur tel qu'une liste, on peut donner l'illusion de renvoyer plusieurs valeurs. Par exemple :

```

1 def f(x, y):
2     return [x + y , x - y]
3
4 L = f(5, 2)
5 print(L)
6 print(L[0]+L[1])
7 [7, 3]
8 10

```

— Ligne 2 : la fonction `f` renvoie un seul objet, à savoir une liste

— Lignes 4 et 6 : une liste ayant plusieurs éléments, on peut avoir l'illusion que la fonction `f` a renvoyé deux valeurs.

c) Instructions `return` multiples

La définition d'une fonction peut contenir des instructions `return` en plusieurs endroits.

Par exemple, soit la fonction `f` qui prend un entier `x` en paramètre et renvoie $x/2$ (la moitié de x) si x est pair et renvoie $3x + 1$ si x est impair. Par exemple $f(5) = 16$ ou encore $f(20) = 10$.

On peut coder cette fonction de la manière suivante :

```

1 def f(x):
2     if x%2 ==0:
3         return x//2
4     else:
5         return 3*x+1
6
7 print(f(5))
8 print(f(20))
9 16
10 10

```

La définition de cette fonction utilise deux instructions `return` : la fin de l'exécution de la fonction se produira, selon les cas, suivant la première instruction `return` (ligne 3) ou suivant la seconde instruction `return` (ligne 5). Bien sûr, un appel de fonction n'exécute jamais

deux instructions `return` l'une à la suite de l'autre puisque toute instruction `return` achève définitivement l'exécution d'une fonction.

La programmation « structurée »

Une certaine conception de la programmation, dite « programmation structurée » recommande de ne pas placer plusieurs instructions `return` dans une fonction parce que cela rendrait plus difficile la compréhension de l'exécution de la fonction puisqu'elle admet plusieurs « points de sortie »². Le code conforme aux principes de la programmation structurée serait plutôt le suivant :

```
1 def f(x):
2     if x%2 ==0:
3         y = x//2
4     else:
5         y = 3*x+1
6     return y
7
8 print(f(5))
9 print(f(20))
```

— La définition de `f` ne comprend qu'un seul `return` : la fonction ne peut terminer son exécution avant d'atteindre la ligne 6.

d) Absence de `return`

Une fonction agit sur des données et peut retourner une valeur explicitement :

avec_return.py

```
1 def f(x):
2     return 100*x
3
4 y=f(42)
5 print(y)
6 4200
```

Cependant, une fonction ne comporte pas nécessairement une instruction `return`. Voici un exemple :

```
1 def g(x):
2     print(100 * x)
3
4 g(42)
5 4200
```

- Lignes 1-2 : la fonction `g` ne contient pas d'instruction `return`. Par contre, elle contient un appel à la fonction `print` qui exprime que l'action de `g` consiste à afficher quelque chose.
- Ligne 4 : la fonction `g` est appelée mais `g(42)` ne renvoie rien. Noter qu'il n'y a pas d'appel à la fonction `print` à cette ligne.
- Ligne 5 : cet affichage n'est pas la valeur de `g(42)` mais le résultat de l'action de l'appel `g(42)` : un affichage de la valeur de `100 * 42`.

2. La validité de ce principe est controversée.

Dans une terminologie aujourd'hui un peu désuète, une fonction qui ne retourne pas de valeur est appelée une *procédure*.

e) Retour d'une fonction sans instruction `return`

La fonction suivante ne possède aucune instruction `return` explicite :

```
1 def g(x):
2     print(100 * x)
3
4 y = g(42)
5 print(y)
6
7 4200
8 None
```

Toutefois, une telle fonction renvoie quelque chose : l'objet `None`. Le code ci-dessus le confirme : la variable `y` référence la valeur de retour de l'appel à fonction `g` et comme l'affichage le montre, cette valeur vaut `None`.

`None` est un mot-clé du langage Python et dont la signification dépend de son utilisation. Il est utilisé pour signifier l'absence de résultat, ou le fait que le résultat doit être considéré comme indéterminé.

L'exemple précédent n'a d'intérêt que pour comprendre qu'une fonction sans `return` renvoie néanmoins quelque chose. Toutefois, la plupart du temps, il n'y a aucun sens à afficher le retour d'une fonction qui ne renvoie rien. Dans le cas de la fonction ci-dessus, une version correcte serait :

```
1 def g(x):
2     print(100 * x)
3
4 g(42)
```

— Ligne 4 : on appelle `g` mais on n'affiche pas la valeur de retour de `g`.

f) Procédure ou pas ?

Si une fonction a pour tâche exclusive d'afficher, on utilisera une procédure autrement dit `f` ne renverra rien et se contentera d'afficher avec la fonction `print`.

Par exemple, si un programme doit déterminer si deux notes sur 20 données conduisent ou pas à une moyenne supérieure à 10, il serait possible d'écrire une fonction comme ceci :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     if ok:
5         print("Moyenne >= 10")
6     else:
7         print("Moyenne < 10")
8
9
10 f(14, 8)
11
12 Moyenne >= 10
```

Toutefois, ce type d'usage doit rester rare.

L'inconvénient de la fonction `f` est qu'elle ne renvoie rien, elle se contente d'afficher, donc c'est utile uniquement pour un humain qui lit la zone d'affichage. Le message affiché ne peut être exploité par un autre programme Python alors que, pourtant, la fonction `f` a fait l'essentiel de la tâche, à savoir :

- le calcul de la moyenne (ligne 2)
- sa comparaison avec la note de 10 (ligne 3).

Pourtant, ces calculs demeurent inaccessibles à l'*extérieur* de la fonction, car la fonction ne renvoie rien.

Il est possible de remédier à cette difficulté en créant *deux* fonctions :

- une fonction `f` qui regarde si, oui ou non, la moyenne est supérieure à 10 et *renvoie* cette comparaison ;
- une procédure qui se contente d'afficher de façon interprétable pour un humain, le résultat renvoyé par `f`.

D'où le programme préférable :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 def afficher(plus_que_10):
7     if plus_que_10:
8         print("Moyenne >= 10")
9     else:
10        print("Moyenne < 10")
11
12 resultat = f(14, 8)
13 afficher(resultat)
14 Moyenne >= 10
```

- Ligne 1 : fonction qui renvoie le résultat de la comparaison avec 10 de la moyenne des notes `a` et `b`.
- Ligne 6 : fonction qui affiche un message concernant un résultat (placé dans la variable `plus_que_10`).

g) Notion d'effet de bord

Soit la fonction suivante :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6 resultat = f(14, 8)
7 print(resultat)
```

8 True

Cette fonction agit de la manière suivante : elle reçoit des arguments et elle renvoie une valeur. Son action est uniquement déterminée par ce qu'elle renvoie. Toutefois, certaines fonctions ont une action qui ne se limite pas à renvoyer une valeur. L'exemple le plus simple est le cas d'un affichage :

```
1 def f(x):  
2     y=10*x  
3     print(y+1)  
4     return y  
5  
6 z=f(42)  
7 print(z)
```

8 421

9 420

La fonction ci-dessus renvoie un objet mais accessoirement a une autre action : l'affichage d'une certaine valeur. Cet action qui n'est pas l'action de renvoyer un objet est appelé un *effet de bord*. En pratique, un effet de bord peut prendre des formes très variées comme un affichage, la création d'un fichier ou la modification d'objets « extérieurs » à la fonction.

3 Créer, utiliser des fonctions

a) Des instructions à la construction d'une fonction

Soit à écrire le code d'une fonction qui calcule le plus grand de deux nombres a et b donnés.

On va exposer le processus de transformation permettant de passer d'un code qui répond au problème mais qui n'utilise pas de fonction à un code équivalent mais placé dans une fonction.

On commence par écrire un code n'utilisant pas de fonction mais qui réponde à la question, quel que soit le choix pour a et b :

```
1 a = 81  
2 b = 31  
3  
4 if a >= b:  
5     print(a)  
6 else:  
7     print(b)
```

8 81

Il est ESSENTIEL d'utiliser des variables car ce sont elles qui vont devenir les paramètres de la fonction à construire.

On remarque que, en l'état :

- le code ne peut être exécuté que pour deux choix particuliers de a et de b , cf. lignes 1-2;
- le code ne place pas le maximum dans un variable mais se contente de l'afficher.

Pour faciliter la conversion du code vers celui d'une fonction qui **renvoie** le maximum entre a et b , on peut placer le maximum dans une variable, par exemple m :

```

maximum_sans_fonction.py
1 a = 81
2 b = 31
3
4 if a >= b:
5     m = a
6 else:
7     m = b
8
9 print(m)
10 81

```

On va maintenant « convertir » ce code en celui d'une fonction. La fonction admet pour paramètres *a* et *b* et elle doit renvoyer le maximum *m*, donc le schéma de la fonction, que l'on va appeler *f*, est le suivant :

```

1 def f(a, b):
2     # code inconnu ...
3     # ...
4     return m

```

Le code inconnu est obtenu en regardant le code `maximum_sans_fonction.py` calculant la maximum *m*. Voici les deux codes comparés :

```

maximum_sans_fonction.py
1 a = 81
2 b = 31
3
4 if a >= b:
5     m = a
6 else:
7     m = b
8
9 print(m)

maximum_fonction.py
10 def f(a, b):
11     if a >= b:
12         m = a
13     else:
14         m = b
15     return m
16
17 a = 81
18 b = 31
19 print(f(a, b))

```

- Lignes 1-2 : ces deux instructions ne servent pas dans le corps de *f*. Ces instructions qui définissaient *a* et *b* sont remplacées par l'en-tête de *f* ligne 10.
- Lignes 4-7 : ces lignes sont préservées à l'identique dans le code de *f*.
- Ligne 9 : comme *f* doit *renvoyer* le maximum *m*, on a remplacé l'affichage par une instruction **return** (ligne 15).
- Lignes 17-19 : on teste *f* de la même façon que le code initial avait été exécuté pour deux

valeurs de **a** et **b**, cf. lignes 1-2.

b) Enchaîner des fonctions

Il est assez fréquent que, dans un programme Python, une fonction fasse appel à une autre fonction qui elle-même fait appel à d'autres fonctions et ainsi de suite. On dit qu'on *enchaîne* les fonctions.

Par exemple, soit le code suivant :

enchaîner_fonctions.py

```
1 def f(x):
2     return g(x)*10
3
4 def g(x):
5     z=h(x) + k(x)
6     print("g : ", z)
7     return z
8
9 def h(x):
10    z=x+2
11    print("h : ", z)
12    return z
13
14 def k(x):
15    z=x+3
16    print("k : ", z)
17    return z
18
19 a = 42
20 print(a, f(a))
21
22 h : 44
23 k : 45
24 g : 89
25 42 890
```

- Ligne 1 : une fonction **f** est définie et fait appel (au sens d'un *appel* de fonction) à une autre fonction **g**.
- Ligne 4 : la fonction **g** est définie et fait elle-même appel à deux autres fonctions, définies plus bas, lignes 9 et 14.
- Lignes 6, 11 et 16 : on a placé des instructions d'affichage pour mieux suivre l'enchaînement des appels des différentes fonctions.
- Ligne 20 : l'appel **f(42)** est lancé.
 - Le code lignes 1-2 est exécuté mais la ligne 2 appelle la fonction **g**. Tant que la fonction **g** appelée ligne 2 n'aura pas renvoyé son résultat (la valeur de **g(x)**), la fonction **f** ne pourra rien renvoyer à l'expression appelante ligne 20 : on dit que la fonction **f** est *en attente*.
 - La fonction **g** appelle elle-même la fonction **h** et la fonction **k**.
 - Lorsque ces fonctions ont renvoyé leur résultat, la somme de ces résultats est placé dans la variable **z** de la définition de **g**. Puis **g** peut renvoyer son résultat à la fonction appelante,

ici, la fonction `f`, à la ligne 2, qui elle-même peut renvoyer son résultat à l'appelant, ici la fonction `print` ligne 20.

Pile d'appels

On reprend le code `enchaîner_fonctions.py` :

`enchaîner_fonctions.py`

```
1 def f(x):
2     return g(x)*10
3
4 def g(x):
5     z=h(x) + k(x)
6     print("g : ", z)
7     return z
8
9 def h(x):
10    z=x+2
11    print("h : ", z)
12    return z
13
14 def k(x):
15    z=x+3
16    print("k : ", z)
17    return z
18
19 print(f(42))
```

Les appels de fonctions s'empilent pour former ce qu'on appelle la *pile* d'appels qu'on imagine comme des appels qui s'empilent de bas en haut.

- Lignes 1-18 : la pile d'appels demeure vide lors de la première exécution de toutes ces lignes.
- Ligne 19 : la fonction `f` est appelée, et donc la pile contient l'appel de `f`
- Ligne 2 : `g` est appelée, donc la pile contient de bas en haut l'appel de `f` et au-dessus l'appel de `g`
- Ligne 5 : la fonction `h` est appelée, la pile contient de bas en haut : les appels de `f` puis `g` puis `h`.
- Ligne 5 : quand `h` renvoie sa valeur (lignes 12) à la fonction `g` qui l'a appelée, la fonction `h` quitte la pile d'appels. On dit alors que `h` *dépille*. La pile contient alors l'appel de `f` puis l'appel de `g` qui sont en attente.
- Ligne 5 : la fonction `k` est appelée donc la pile est constituée de bas en haut de : l'appel de `f`, l'appel de `g`, l'appel de `k`.
- `k` dépille (lignes 17) puis quand `g` renvoie `z` (lignes 7) à `f` (ligne 2) qui dépille aussi et la pile est à nouveau vide.

La pile des appels est bien visible à l'aide d'un débogueur.

c) De l'usage des fonctions

Pour un débutant qui a déjà écrit du code Python mais sans utiliser encore la notion de fonction, écrire une fonction peut être une tâche difficile car il s'agit en général de transposer en code un raisonnement.

Intérêt d'écrire des fonctions

L'utilisation de fonctions présente les trois intérêts suivants :

- le code est plus **lisible** que s'il n'est pas enveloppé dans une fonction car une fonction identifie clairement :
 - une tâche (virtuelle) à exécuter avec une interface constituée de paramètres que l'on donne à la fonction et d'un résultat (ce que renvoie l'instruction **return**),
 - d'un (ou plusieurs) appel(s) à la fonction ;
- le code est **réutilisable**. Si on n'écrit pas le code dans une fonction, il faudrait réécrire tout le code correspondant chaque fois que l'on souhaiterait effectuer la même tâche ;
- le code est plus économique. Appeler une fonction, c'est une ligne de code qui en fait en appelle souvent plusieurs dizaines puisque l'appel équivaut à l'exécution de toutes les lignes de code de la fonction et des autres fonctions que la fonction appelle elle-même.

Découpage en tâches

Pour mieux organiser son code, le programmeur cherche souvent à découper la tâche à effectuer en plusieurs fonctions, chacune réalisant une certaine sous-tâche (on dit parfois « service » au lieu de « tâche » ou « sous-tâche »).

Lorsqu'on est débutant avec les fonctions, il n'est pas toujours facile d'arriver à découper son programme en fonctions pertinentes.

Lors de l'écriture d'une fonction **f**, le programmeur doit d'abord s'interroger sur l'*interface* de la fonction **f** :

- quels seront les paramètres de **f**? autrement dit, de quelles données la fonction **f** a-t-elle besoin ?
- qu'est ce que la fonction **f** doit renvoyer ?
- la fonction a-t-elle des effets de bords ?

Ensuite, il doit écrire le code de définition de la fonction **f** qui va être capable de réaliser la tâche : c'est ce qu'on appelle l'*implémentation*³ de la fonction.

Exemple

Par exemple, soit le tableau de Pascal :

1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

3. Terme à préférer à celui-d'*implantation* parfois utilisé mais à tort,

Chaque élément z d'une ligne s'obtient comme somme des deux termes au-dessus et à gauche de l'élément z . Par exemple, à la ligne 7, on a : $35 = 20 + 15$. On veut écrire un programme qui affiche le tableau de Pascal jusqu'à sa n^e ligne .

Plus précisément, voici les règles de construction du Tableau de Pascal. Le Tableau de Pascal est une suite de lignes telles que :

- la 1^{er} ligne est 1 1 ;
- chaque ligne commence et se termine par le nombre 1 ;
- pour chaque élément z qui ne figure pas aux extrémités d'une ligne L du tableau, on a $z = x + y$ où x est le terme sur la ligne précédente de L et situé au-dessus de z et y est le voisin de gauche de x .

Objectif : écrire une fonction `pascal(n)` qui prend en paramètre un numéro n de ligne et qui affiche les n premières lignes du tableau de Pascal.

Pour afficher une ligne, il suffit de connaître la précédente (disons L) et d'appliquer les règles de construction du Tableau de Pascal. Cette tâche sera accomplie par une fonction `ligne_suivante(L)`.

Voici un code possible de la fonction `ligne_suivante(L)`, accompagné d'un test :

```

1 def ligne_suivante(L):
2     LL=[1]
3     n=len(L)
4     for k in range(1, n):
5         LL.append(L[k-1]+L[k])
6     LL.append(1)
7     return LL
8
9 L=[1, 3, 3, 1]
10 print(ligne_suivante(L))
11 [1, 4, 6, 4, 1]
```

- Ligne 2 : `LL` est la future ligne suivante et au départ une liste contenant juste le 1 initial de chaque ligne et qui va être remplie au fur et à mesure.
- Lignes 6 : l'autre extrémité de chaque ligne du tableau vaut 1.
- Lignes 4-5 : la règle de construction du terme `LL[k]` de la ligne suivante de `L` dans le Tableau de Pascal. On sait que `LL[k]` est la somme des deux termes de `L` situés en haut et à gauche de `LL[k]`.
- Ligne 7 : la fonction doit renvoyer la nouvelle ligne.
- Lignes 9-11 : un test, positif puisque la ligne suivante de la ligne 1 3 3 1 est bien 1 4 6 4 1.

La fonction `pascal(n)` va utiliser la fonction `ligne_suivante(L)` dans une boucle pour donner une première version du tableau de Pascal que l'on va affiner ensuite :

```

1 def ligne_suivante(L):
2     LL=[]
3     LL.append(1)
4     n=len(L)
5     for k in range(1, n):
6         LL.append(L[k-1]+L[k])
7     LL.append(1)
8     return LL
9
10 def pascal(n):
11     L=[1, 1]
12     print(L)
13     for i in range(0, n-1):
14         LL=ligne_suivante(L)
15         L=LL
16         print(L)
17
18 pascal(10)

```

```

19 [1, 1]
20 [1, 2, 1]
21 [1, 3, 3, 1]
22 [1, 4, 6, 4, 1]
23 [1, 5, 10, 10, 5, 1]
24 [1, 6, 15, 20, 15, 6, 1]
25 [1, 7, 21, 35, 35, 21, 7, 1]
26 [1, 8, 28, 56, 70, 56, 28, 8, 1]
27 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
28 [1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

```

- Ligne 10 : il y a n lignes à afficher
- Lignes 11-12 et 19 : on affiche la 1^{er} ligne du Tableau de Pascal
- Ligne 13 : il reste $n-1$ lignes à afficher d'où le `range(n-1)`.
- Ligne 14 : L est la ligne courante, LL la ligne suivante, renvoyée par la fonction `ligne_suivante`
- Ligne 15 : le point-clé du programme : on prépare le tour suivant dans la boucle en mettant à jour la ligne courante. Il aurait été possible de remplacer le code des lignes 14-15 par l'unique ligne `L=ligne_suivante(L)`.
- Ligne 16 : affichage de la ligne courante.
- Lignes 18 et 19-28 : on reconnaît bien les 10 premières lignes du Tableau de Pascal.

On se rend compte en regardant l'affichage obtenu qu'il peut être amélioré pour faire afficher juste des nombres, sans virgules ni crochets. Pour cela, il suffit d'écrire une fonction `afficher_ligne` dont la tâche est d'afficher le contenu d'une liste sur une seule ligne en séparant les éléments par une seule espace :

```

1 def afficher_ligne(L):
2     for i in range(len(L)):
3         print(L[i], end = " ")
4     print()
5
6 afficher_ligne([1, 4, 6, 4, 1])
7 afficher_ligne([1, 8, 28, 56, 70, 56, 28, 8, 1])

```

```

8 1 4 6 4 1
9 1 8 28 56 70 56 28 8 1

```

Il ne reste plus qu'à remplacer l'affichage `print(L)` de la fonction `pascal(n)` à la ligne 16 par `afficher_ligne(L)` pour obtenir le programme attendu :

```

1 def ligne_suivante(L):
2     LL=[1]
3     n=len(L)
4     for k in range(1, n):
5         LL.append(L[k-1]+L[k])
6     LL.append(1)
7     return LL
8
9 def pascal(n):
10    L=[1, 1]
11    afficher_ligne(L)
12    for i in range(0, n-1):
13        L=ligne_suivante(L)
14        afficher_ligne(L)
15
16 def afficher_ligne(L):
17     for i in range(0, len(L)):
18         print(L[i], end = " ")
19     print()
20
21 pascal(10)

```

```

22 1 1
23 1 2 1
24 1 3 3 1
25 1 4 6 4 1
26 1 5 10 10 5 1
27 1 6 15 20 15 6 1
28 1 7 21 35 35 21 7 1
29 1 8 28 56 70 56 28 8 1
30 1 9 36 84 126 126 84 36 9 1
31 1 10 45 120 210 252 210 120 45 10 1

```

L'affichage pourrait être encore amélioré pour faire apparaître des colonnes bien alignées.

4 Divers

a) Fonction sans paramètre

Il est possible définir des fonctions n'admettant pas de paramètre, comme la fonction `f` ci-dessous (ligne 1) :

```
1 def f():
2     return 42
3
4 print(f())
5 42
```

Pour appeler la fonction `f`, on écrit (ligne 4) juste `f()` autrement dit, on place juste une paire de parenthèses vides.

En pratique, les fonctions sans paramètre sont beaucoup moins fréquentes que leurs analogues avec paramètres. Mais les fonctions sans paramètre servent parfois à regrouper des instructions dans une fonction pour les appeler en une seule fois, ce qui permet d'avoir un code plus structuré.

Un cas typique serait une fonction utilisée pour tester d'autres fonctions :

```
1 def f(a, b):
2     m = (a + b)/2
3     ok = (m >= 10)
4     return ok
5
6
7 def tests():
8     resultat = f(14, 8)
9     print(resultat)
10    resultat = f(0, 20)
11    print(resultat)
12    resultat = f(5, 6)
13    print(resultat)
14
15 tests()
16 True
17 True
18 False
```

- La fonction `tests` n'utilise aucune donnée et donc n'admet aucun paramètre.
- La fonction `tests` n'admettant aucun paramètre, elle n'admet non plus aucun argument quand elle est appelée.

L'intérêt est qu'il suffit de retirer l'appel à `tests` pour désactiver le code qui est dans la fonction `tests`, ce qui n'était pas faisable facilement si le code était placé hors d'une fonction.

Autre exemple

Soit à coder une fonction `lancer_un_de` simulant le jet d'un dé à 6 faces : la fonction renvoie un entier au hasard entre 1 et 6. Une telle fonction ne dépend de rien et donc sera définie sans paramètre, plus précisément :

```

1 from random import randrange
2
3 def lancer_un_de():
4     return randrange(1,7)
5
6 jet=lancer_un_de()
7 print(jet)
8 5

```

- Lignes 3-4 : fonction sans paramètre.
- Ligne 6 : appel sans argument de la fonction.

b) Variables locales

La notion de variable locale

variable_locale.py

```

1 def f(x):
2     y=x+1
3     return 10*y
4
5 print(f(8))

```

Lorsqu'on définit une fonction, ici la fonction `f` lignes 1-3, les variables dites *locales* à la fonction `f` sont

- les variables données en paramètre, comme `x` (ligne 1)
- les variables **créées** dans la définition de la fonction, par exemple `y` à la ligne 2.

Les objets que représentent ces variables n'existent que pendant l'exécution de la fonction `f`, c'est-à-dire uniquement lors de l'appel de la fonction `f`. Ainsi, la variable `x` n'a pas d'existence tant que `f` n'a pas été appelée. En outre, après la fin de l'exécution de la fonction, la variable `x` n'a plus d'existence non seulement en mémoire mais aussi dans le code-source. Les variables locales disparaissent avec l'exécution de la fonction.

Tentative d'accès à une variable locale

Le caractère local de variables comme `x` ou `y` signifie que ces variables ne sont pas connues à l'extérieur du code de la fonction `f`. Si on tente d'y accéder, on obtient une erreur.

Voici deux codes qui illustrent des tentatives de lecture de variables locales :

```

1 def f(x):
2     y=x+1
3     return 10*y
4
5 print(f(8))
6 print(x)

```

```

7 90
8 Traceback (most recent call last):
9   File "a.py", line 6, in <module>
10     print(x)
11 NameError: name 'x' is not defined

```

- Lignes 5 et 1 : la variable `x` prend la valeur 8 pendant l'exécution de `f` pendant l'appel ligne 5.
- Ligne 11 : pourtant, la variable `x` N'est PAS reconnue en dehors du code de définition de la fonction `f`.
- Ligne 7 : le résultat de l'affichage de la ligne 5.

```

1 def f(x):
2     y=x+42
3     z=10
4     return 2*y+z
5
6 print(f(8))
7 print(z)

```

```

8 110
9 Traceback (most recent call last):
10   File "a.py", line 7, in <module>
11     print(z)
12 NameError: name 'z' is not defined

```

- Ligne 3 : définition de la variable locale `z` de la fonction `f`.
- Ligne 6 : pendant l'appel de `f`, la variable `z` est utilisée
- Lignes 7 et 12 : après l'appel de `f`, la variable `z` cesse complètement d'exister.

Remède

Une variable locale à une fonction n'a pas vocation à être accédée depuis l'extérieur de la fonction. Si on veut accéder à une variable locale (en fait à son contenu), il faut que la fonction renvoie le contenu de cette variable locale ou renvoie un objet qui permette d'accéder à la valeur de cette variable locale.

c) Variables globales

Soit le code suivant :

```

1 def f(x):
2     print(z)
3     return z + x
4
5 z=42
6
7 print(f(8))

```

```

8 42
9 50

```

La variable `z` déclarée à la ligne 5 en dehors de toute fonction : on dit que `z` est une variable *globale*. La fonction `f` définie à la ligne 1 a accès à cette variable (lignes 2-3). Observons que :

- la variable globale `z` est définie **après** la définition de `f`. En fait ce qui compte, c'est que `z` soit définie **avant l'appel** (ligne 7) à la fonction `f`.
- l'accès à la variable globale `z` se fait en **lecture** et non **en écriture** ie la variable `z` n'est pas modifiée par la fonction `f`.

Quand on débute en programmation, et qu'on commence à manipuler des fonctions, il est conseillé d'éviter d'utiliser des variables globales. L'intérêt d'une fonction est de constituer un environnement autonome d'exécution : en principe, une fonction ne doit dépendre d'aucun élément extérieur autre que des appels éventuels à d'autres fonctions⁴.

Si, pour des raisons exceptionnelles, une fonction est amenée à utiliser une variable globale, il est souhaitable, pour des raisons de lisibilité, de déclarer ces variables en début de la zone d'édition du code source, avant les fonctions qui y font appel.

d) Constantes placées en variables globales

Limiter l'usage de variables globales est considéré comme une bonne pratique de programmation. Utiliser des constantes en variables globales est un cas toléré. Il s'agit de variables qui référencent des objets qui ne changeront pas durant toute la vie du programme.

Un programme utilisant une valeur de π pour calculer des aires ou des volumes pourra définir π comme variable globale. Voici un exemple :

```
1 PI = 3.1415926
2
3 def disque(r):
4     return PI * r * r
5
6 def sphere(r):
7     return 4 * PI * r * r
8
9 print(disque(10))
10 print(sphere(10))
11 314.15926
12 1256.63704
```

— Ligne 1 : constante déclarée en variable globale.

— Lignes 4 et 7 : utilisation de PI dans des fonctions.

L'usage veut que des variables globales référençant des constantes soient écrites en début de fichier et en majuscules.

e) Fonction non appelée

Soit le code suivant :

```
1 z=10
2
3 def f(x):
4     print(10 * x)
5
6 print(z+1)
7 11
```

— Lignes 3-4 : une fonction **f** est définie mais cette fonction n'est appelée nulle part dans le programme.

4. Certains programmeurs vont même jusqu'à refuser d'écrire tout code exécutable autre que dans une fonction.

En pratique, un code n'a pas de raison de définir une fonction sans appeler cette fonction avec des arguments. Toutefois, cela signifie que vous pouvez, sans danger, écrire la définition d'une fonction qui, par exemple, serait inachevée, ou qui ne marcherait pas encore ou qui ne serait qu'une ébauche ou un template d'un programme à exécuter ultérieurement.

Mais il existe des raisons plus sérieuses à écrire des fonctions sans les appeler dans le fichier où les fonctions sont définies : il est possible d'appeler ces fonctions depuis un autre fichier Python.

f) Le passage des arguments par affectation

Soit le code :

```
1 def f(d,u):
2     N = 10 * d + u
3     return N
4
5 print(f(4,2))
6 42
```

— Une définition de fonction

— Un appel de la fonction avec les arguments 4 et 2

Lors de l'appel `f(4,2)` ligne XX dans le code ci-dessus, la transmission des arguments 4 et 2 aux paramètres de `f` est effectuée **exactement** comme dans le code ci-dessous :

```
1 d = 4
2 u = 2
3
4 N = 10 * d + u
5
6 print(N)
```

Le lien qui existe entre un paramètre, par exemple `d`, son argument 4, est une affectation (ligne XX), d'où la terminologie, propre à Python de *passage des arguments par affectation*. L'objet que le paramètre `d` reçoit au moment de l'exécution de la fonction `f` est exactement le même objet que l'argument. Non seulement, l'objet reçu a la même valeur, mais, mieux, ils sont identiques.

Pour se rendre compte qu'il s'agit du même objet, on utilise la fonction standard `id` qui permet d'identifier un objet en renvoyant son « identifiant » unique afin de le discerner d'autres objets :

```
1 def f(d,u):
2     print("id(d) ->", id(d))
3     N = 10 * d + u
4     return N
5 x=4
6 print("id(x) ->", id(x))
7
8 print(f(4,2))
9 id(x) -> 137220912
10 id(d) -> 137220912
11 42
```

— On affiche l'id de l'objet associé à la variable `x`, ici l'objet 4.

- On a modifié `f` pour qu'elle affiche l'id de l'objet qu'elle reçoit pour le paramètre `d`
- On constate qu'il s'agit des mêmes objets.

En particulier, le passage des arguments lors de l'appel d'une fonction se fait sans copie de l'objet. Le fait qu'il n'y ait pas de copie est une garantie d'efficacité. Toute action que la fonction en s'exécutant peut avoir sur un objet doit pouvoir être exécutée comme si l'objet était accédé par une affectation.

g) Modification par une fonction d'un de ses arguments

Une fonction peut-elle modifier un objet qu'elle reçoit en argument ? La réponse doit être nuancée en fonction du sens que l'on donne au verbe *modifier*.

En aucun cas, une fonction ne peut substituer un autre objet à un objet qu'elle reçoit en argument.

Examinons une tentative de modification par une fonction d'un objet que la fonction reçoit en argument. Soit par exemple le code ci-dessous

```

1 def f(x):
2     x = 42
3
4 a=10
5 print(a)
6 f(a)
7 print(a)
8 10
9 10

```

- La valeur de `a` avant l'appel
- l'action de `f` qui tente de modifier l'objet reçu en argument
- La valeur de `a` est inchangée et c'est le même objet 10 que celui vers lequel `a` se référait avant l'appel.

La fonction `f` affecte 42 au paramètre `x`. Si on passe en argument à `f` une autre valeur comme 10, cette valeur va-t-elle être remplacée par 42 ? Non, comme le montre l'affichage ci-dessus.

Pour comprendre le comportement de `f` et pourquoi `a` n'est pas modifié, il suffit de se rappeler que l'action d'un passage des arguments en Python est une transmission par affectation et donc le code ci-dessus est équivalent à :

```

1 a=10
2 print(a)
3
4 # ----- Équivalent de l'action f(a)
5 x = a
6 x = 42
7 # -----
8
9 print(a)
10 10
11 10

```

Les affectations `x=a` et `x=42` ne peuvent pas remplacer `a` par `42`; en effet, `x=42` ne fait que créer un nouvel objet `42` et lui associer le nom `x`, autrement dit, la variable `x` ne réfère plus l'objet `10` qu'elle référerait ligne 5. On voit que ces affectations ne peuvent modifier `a` qui lui ne subit aucune affectation.

À défaut de remplacer un objet par un autre, une fonction peut-elle modifier le contenu d'un objet ? La réponse dépend de la nature de l'objet.

Si une fonction reçoit en argument un objet immuable comme un entier ou une chaîne, par définition, l'objet ne pourra pas être modifié, et a fortiori par `f`.

Si une fonction reçoit en argument un objet mutable comme une liste, l'objet pourra être modifié pas la fonction. Voici un exemple où une liste est modifiée par une fonction :

```
1 def f(M):
2     M[0] = 42
3
4 L=[81, 12, 31, 65]
5 print(L)
6 print()
7
8 f(L)
9
10 print(L)
```

```
11 [81, 12, 31, 65]
```

```
12
```

```
13 [42, 12, 31, 65]
```

- `f` est appelée sur une liste d'entiers
- l'action de `f` consiste à remplacer le premier élément de `M` par `42`
- une fois l'appel de `f` terminé, on constate que l'objet `L` placé en paramètre a été modifié.

Une fois de plus, le mode de passage des arguments par affectation propre à Python permet de prévoir ce comportement. En effet le code ci-dessus est équivalent à

```
1 L=[81, 12, 31, 65]
2 print(L)
3 print()
4
5 # ---- Début de l'action de f
6
7 M=L
8 M[0]=42
9
10 # ---- Fin de l'action de f
11
12 print(L)
```

```
13 [81, 12, 31, 65]
```

```
14
```

```
15 [42, 12, 31, 65]
```

L'exemple ci-dessus ne fait pas croire que l'argument a été changé : après appel, `L` référence toujours le même objet. La différence, c'est que le contenu de `L` a changé.

h) Boucle for et return

Le corps d'une boucle peut contenir n'importe quelle instruction ; en particulier, si une boucle for apparaît dans la définition d'une fonction, elle peut être interrompue par une instruction return.

Par exemple, soit à définir une fonction qui prend une liste L en paramètre et renvoie True si L ne contient aucun entier strictement négatif et False sinon :

```
1 def f(L):
2     for i in range(len(L)):
3         if L[i] < 0:
4             return False
5     return True
6
7 L= [31, 82, -42, 32, -81]
8 print(L, f(L))
9
10 print("=====")
11
12 L= [31, 82, 421, 32, 81]
13 print(L, f(L))
14 [31, 82, -42, 32, -81] False
15 =====
16 [31, 82, 421, 32, 81] True
```

- Lignes 1-5 : la fonction f répond au problème posé.
- Ligne 4 : une instruction return interrompt l'exécution de la fonction.
- Lignes 3-4 : la liste L est parcourue et si un terme de la liste L est un nombre négatif, le parcours de la liste ainsi que l'exécution de la fonction sont interrompus et la fonction renvoie False
- Ligne 5 : si la liste L est parcourue sans que jamais le parcours ne soit interrompu par la détection d'un nombre négatif dans la liste, la fonction f se termine en renvoyant True puisque tous les termes de L sont positifs ou nuls.

Résumons le schéma ci-dessus :

- placement d'une boucle for dans une fonction ;
- parcours avec la boucle for d'une liste jusqu'à ce qu'une certaine condition soit vérifiée ;
- interruption de la boucle et de la fonction par une instruction return renvoyant un booléen.

L'intérêt de ce schéma est que le parcours de la liste est optimal : le parcours est interrompu dès que la réponse est connue, ce qui n'est pas le cas de la méthode utilisant un simple drapeau.

i) Le Hasard en Python

Un nombre *aléatoire* est un nombre *tiré au hasard*. Plus généralement, l'adjectif *aléatoire* se réfère à ce qui est le résultat du *hasard*.

Le module standard `random` gère l'aléatoire en Python.

Le code ci-dessous génère un entier aléatoire entre 1 et 49, comme pour un tirage d'une boule au loto :

```

1 from random import randint
2
3 a = randint(1,49)
4 print(a)
5 42

```

- Ligne 1 : la bibliothèque standard Python dispose d'un module de création d'objets aléatoires. Ce module s'appelle `random`. Le module `random` « possède » une fonction `randint`. La ligne 1 permet au programme de faire appel à la fonction `randint`.
- Ligne 3 : le tirage est effectué grâce à la fonction `randint`. La fonction `randint` renvoie ici un nombre (aléatoire) entre 1 et 49, bornes comprises.
- Lignes 4 et 5 : le résultat du tirage aléatoire.

j) Fonction `randrange`

Dans l'exemple ci-dessous, on effectue un tirage aléatoire de 6 entiers n vérifiant $1 \leq n < 50$ (comme au loto en France) :

```

1 from random import randrange
2
3 print(randrange(1, 50))
4 print(randrange(1, 50))
5 print(randrange(1, 50))
6 print(randrange(1, 50))
7 print(randrange(1, 50))
8 print(randrange(1, 50))
9 14
10 15
11 47
12 45
13 38
14 12

```

La fonction `randrange` est une fonction du module `random`. L'usage de la fonction `randrange` nécessite donc une instruction d'importation, cf. ligne 1 ci-dessus.

Un appel `randrange(a, b)` renvoie un entier aléatoire n tel que $a \leq n < b$ où a et b ont des valeurs entières (on notera que la borne en b est *stricte*)

Pour effectuer un tirage par pile ou face, on utilisera `randrange(2)`.

De même, pour simuler un dé, on utilisera `randrange(6)` :

```

1 from random import randrange
2
3 # Tirage de dé à 6 faces
4 de_1_6 = 1+randrange(6)
5 print(de_1_6)
6 3

```

k) Fonction `randint`

La fonction `randint` fournie par le module standard `random` génère un entier aléatoire entre deux bornes entières `a` et `b` qui elles-même peuvent être générées.

```
1 from random import randint
2
3 print(randint(1, 49))
```

```
4 40
```

Un appel `randint(a, b)` se fait toujours avec *deux* arguments, entiers et renvoie un entier aléatoire `n` tel que $a \leq n \leq b$.

Pour effectuer un tirage par pile ou face, on utilisera `randint(1, 2)`.

De même, pour simuler un dé, on utilisera `randint(1,6)` :

```
1 from random import randint
2
3 # Tirage de dé à 6 faces
4 de_1_6 = randint(1, 6)
5 print(de_1_6)
```

```
6 3
```

5 Exercices

EXERCICES TYPE

Fonction aire de disque

Écrire une fonction `aire_disque` qui prend en paramètre un entier `r` et renvoie la surface du disque de rayon r , définie par $S = \pi r^2$. On prendra 3.14 comme valeur approchée de π .

Solution

Le code ne présente pas de difficulté, il s'agit d'implémenter une simple formule. Seule la manière de présenter π peut poser question.

On peut toujours écrire :

```
1 def aire_disque(r):
2     return 3.14 * r**2
3
4 print(aire_disque(10))
```

```
5 314.0
```

Il est plus lisible d'utiliser un nom de variable. Voici deux variantes :

```

1 PI = 3.14
2
3 def aire_disque(r):
4     return PI * r**2
5
6 print(aire_disque(10))

```

```

7 def aire_disque(r):
8     PI = 3.14
9     return PI * r**2
10
11 print(aire_disque(10))

```

Il est d'usage d'écrire des variables qui représentent des constantes en lettres capitales.

Il est aussi possible pour obtenir des calculs plus précis de demander au module `math` de Python de fournir une valeur approchée de π :

```

1 from math import pi
2
3 def aire_disque(r):
4     return pi * r**2
5
6 print(aire_disque(10))

```

```

7 314.1592653589793

```

```

2 from math import pi
3
4 def aire_disque(r):
5     return pi * r**2
6
7 print(aire_disque(10))

```

```

7 PI=3.14
8 def surface(r):
9     return PI*r**2
10 print(surface(10))

```

Implémenter la fonction *signe*

Écrire une fonction `signe` qui évalue le signe d'un paramètre `x` :

$$\text{signe}(x) = \begin{cases} 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \\ 0 & \text{si } x = 0 \end{cases}$$

Solution

Il faut utiliser des instructions `if/else` :

```

1 def signe(x):
2     if x>0:
3         return 1
4     else:
5         if x<0:
6             return -1
7         else:
8             return 0
9 print(signe(0), signe(4), signe(-7))
10 0 1 -1

```

Il est possible d'alléger un peu le code en se rappelant qu'une instruction **return** interrompt définitivement l'exécution d'une fonction :

```

1 # Alternative
2
3 def signe(x):
4     if x>0:
5         return 1
6     if x<0:
7         return -1
8     return 0
9
10
11 print(signe(0), signe(4), signe(-7))
12 0 1 -1

```

- Ligne 6 : un **else** n'est pas utile car l'instruction **return** au-dessus entraîne que si le code accède à la ligne 6, c'est que forcément $x > 0$ est faux, ce qui correspond à la situation d'un **else**.
- Ligne 8 : pour la même raison, si l'exécution du code arrive à cette ligne, c'est forcément que $x = 0$.

Alternative utilisant **if/elif/else**

Si la clause **elif** est connue, il est préférable d'écrire le code ainsi :

```

1 def signe(x):
2     if x>0:
3         return 1
4     elif x<0:
5         return -1
6     else:
7         return 0
8
9 print(signe(0), signe(4), signe(-7))

```

Il serait possible de simplifier bien davantage en utilisant une « expression conditionnelle ».

Périodes de temps

Dans cet exercice, on va considérer des périodes sous la forme

a années, m mois et j jours

où $0 \leq m < 12$ et $0 \leq j < 30$ où a, m et j sont des entiers. Pour simplifier, on considérera qu'une année dure *toujours* 365 jours et qu'un mois dure *toujours* 30 jours.

- ① Écrire une fonction `amj_to_j` qui retourne le nombre de jours d'une période exprimée en années, mois et jours. Vérifier que 10 ans, 4 mois et 22 jours correspondent à 3792 jours.
- ② *Cette question est indépendante de ce qui précède.* Dans cette question, l'usage de la méthode `append` est inapproprié.
Écrire une fonction `j_to_amj` qui prend en paramètre une période exprimée en jours et la retourne exprimée en années, mois et jours sous forme de liste `[a, m, j]`. Bien sûr, on aura $m < 12$ et $j < 30$. Vérifier que 10000 jours correspondent à 27 ans, 4 mois et 25 jours.
- ③ *Cette question est indépendante de ce qui précède.* Écrire une procédure `afficher_pperiode` qui prend en paramètre une liste de 3 entiers correspondant à une période de a années, m mois et j jours et qui affiche

```
1 a années m mois et j jours
```

Par exemple, `afficher_pperiode([27, 4, 1])` affiche exactement :

27 années 4 mois et 1 jours

- ④ *Votre réponse à cette question doit utiliser les fonctions définies dans les questions précédentes. Cette question ne demande pas de définir de nouvelle fonction.*

Un père a un âge de 42 ans, 4 mois et 2 jours et sa fille a un âge de 9 ans 9 mois et 28 jours. Écrire un programme qui affiche la différence d'âge, exprimée en années, mois et jours, entre le père et sa fille.

Il est attendu que votre code tienne sur une seule ligne.

Solution

- ① D'après les hypothèses simplificatrices de l'énoncé, le nombre de jours dans une période de a années, m mois et j jours est donné par la formule :

$$365*a + 30*m + j$$

d'où le code :

```
1 def amj_to_j(a, m, j):  
2     return 365*a+30*m+j  
3  
4 amj_to_j(10, 4, 22)  
5 3792
```

— Ligne 5 : on retrouve bien le nombre de jours annoncés.

- ② Une période `jours` contient `jours // 365` années entières avec un reliquat de `jours % 365` jours. De la même façon, on convertit ce reliquat de jours en mois et jours. D'où le code :


```

1 def j_to_amj(jours):
2     a=jours//365
3     r=jours%365
4     m=r//30
5     j=r%30
6     return [a, m, j]
7
8 print(j_to_amj(10000))
9 [27, 4, 25]

```

— Ligne 9 : on retrouve bien 27 ans, 4 mois et 25 jours.

- ③ Il s'agit d'écrire une procédure d'affichage. La fonction reçoit un **argument** sous forme de liste `[j, m, a]`. Mais le paramètre ne peut avoir cette forme, un paramètre étant toujours un nom de variable, disons ici `periode`. Dans le code de définition de la fonction, on accède aux années, mois et jours en utilisant un indice puisque `periode` représente une liste de 3 entiers. D'où le code :

```

1 def afficher_periode(periode):
2     print(periode[0], "années", periode[1], "mois", "et", periode[2], "jours")
3
4 afficher_periode([27, 4, 1])
5 27 années 4 mois et 1 jours

```

- ④ Pour calculer un écart entre deux durées, il suffit de
- convertir chaque durée en jours avec la fonction `amj_to_j`
 - de faire la différence des durées en jours
 - de reconvertir cette durée en jours en une période en années, mois et jours avec la fonction `j_to_amj`.

Enfin, il restera à afficher la durée avec la fonction `afficher_periode`.

D'où le code :

```

1 afficher_periode(j_to_amj(amj_to_j(42, 4, 2)-amj_to_j(9, 9, 28)))
2 32 années 6 mois et 9 jours

```

Grille en équerres

- ① Écrire une procédure `f(i, n)` où `i, n` sont des entiers tels que $1 \leq i \leq n$ qui affiche, sur une même ligne, `n` entiers séparés par une espace dont les `i` premiers entiers sont `1, 2, ..., i` et les `n - i` derniers sont tous identiques à l'entier `i`. Par exemple, `f(5, 9)` doit afficher la ligne suivante :

1 2 3 4 5 5 5 5 5

- ② En déduire une procédure `g(n)` qui affiche une grille carrée « en équerres » telle que celle qui figure ci-contre (dans cet exemple `n = 9`). Deux nombres sur une même ligne seront séparés par une espace. Observez bien comment sont placés « en équerres » les nombres 1, 2, etc.

```

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

```

Solution

① Pour calculer $f(i, n)$, il faut effectuer deux actions :

- la génération des entiers de 1 à i ;
- la répétition $n - i$ fois de l'entier i .

Ces deux actions répètent quelque chose et donc se codent chacune avec une boucle **for**. D'où le code :

```

1 def f(i,n):
2     for j in range(0, i):
3         print(j+1, end=' ')
4
5     for k in range(0,n-i):
6         print(i, end = ' ')
7
8 f(5,9)
9 print()
10 print()
11 f(1,9)
12 1 2 3 4 5 5 5 5 5
13
14 1 1 1 1 1 1 1 1 1

```

- Lignes 3 et 6 : pour séparer chaque entier du suivant par juste un espace, on utilise l'argument nommé `end=" "` pour la fonction **print**.

② Pour afficher le motif 2D demandé, il suffit de remarquer que la 1^{re} ligne du motif est $f(1, n)$, la 2^e ligne est $f(2, n)$ et ainsi de suite. Donc, pour afficher le motif, il suffit de répéter avec une boucle **for** l'affichage de chaque ligne. D'où le code :

```

1 def f(i,n):
2     for j in range(0, i):
3         print(j+1, end=' ')
4
5     for k in range(0,n-i):
6         print(i, end = ' ')
7
8 def g(n):
9     for k in range(n):
10        f(k+1,n)
11        print()
12
13 g(9)

```

```

14 1 1 1 1 1 1 1 1 1
15 1 2 2 2 2 2 2 2 2
16 1 2 3 3 3 3 3 3 3
17 1 2 3 4 4 4 4 4 4
18 1 2 3 4 5 5 5 5 5
19 1 2 3 4 5 6 6 6 6
20 1 2 3 4 5 6 7 7 7
21 1 2 3 4 5 6 7 8 8
22 1 2 3 4 5 6 7 8 9

```

EXERCICES

Fonction : définition et appel

Écrire une fonction f définie comme suit : $f(x) = 3x^2 - 1$. Dans le programme principal, vérifier que $f(0) = -1$, $f(1) = 2$ et $f(5) = 74$.

Fonction Celsius -> Fahrenheit

Écrire une fonction $C2F(C)$ qui prend en paramètre une température C (en degrés Celsius) et qui renvoie sa conversion F (en degrés Fahrenheit), suivant la formule

$$F = C * 1,8 + 32$$

Combien font $42^\circ C$ et $-273,15^\circ C$ en $^\circ F$?

Fonction pour année bissextile

Écrire une fonction `bissextile` qui prend en paramètre un entier a et retourne `True` si l'année a est bissextile (`False` sinon). Pour rappel, une année est bissextile si elle est soit multiple de 4 mais pas de 100, soit multiple de 400. Les années 2020 et 2038 sont-elles bissextiles ?

Afficher un mouvement

Soit une liste L formée d'entiers parmi $-1, 0$ ou 1 , par exemple $L = [-1, 0, -1, 0, 0, 1, -1, 0]$. Chaque entier représente une direction de déplacement :

- -1 représente le déplacement à *gauche*
- 0 représente le déplacement *tout droit*
- 1 représente le déplacement à *droite*

Ecrire une fonction `afficherDirection` qui prend en paramètre une liste `L` formée d'entiers parmi `-1`, `0` ou `1` et qui affiche une suite de directions de déplacement.

Par exemple, si `L = [-1, 0, -1, 0, 0, 1, -1, 0]` alors la fonction affichera :

```
1 à gauche
2 tout droit
3 à gauche
4 tout droit
5 tout droit
6 à droite
7 à gauche
8 tout droit
```

L'utilisation d'une liste est encouragée mais il est toutefois possible de répondre à l'exercice sans y avoir recours

Implémenter la fonction `max`

Écrire une fonction `max` qui renvoie le maximum de deux nombres `a` et `b` passés en paramètre.

Fonction binomiale

- ① Écrire une fonction `factorielle` qui renvoie la valeur factorielle de `n` où `n` étant passé en paramètre. Par exemple `1! = 1` et `5! = 120`. On convient que `0! = 1`. On écrira une version de la fonction factorielle utilisant une boucle `for`.
- ② On rappelle que si `n` et `p` sont des entiers positifs ou nuls, le coefficient binomial est défini par

$$\binom{n}{p} = \begin{cases} \frac{n!}{p!(n-p)!} & \text{si } 0 \leq p \leq n \\ 0 & \text{sinon} \end{cases}$$

Ecrire une fonction `binomial(n, p)` qui renvoie le coefficient binomial ci-dessus.

- ③ Ecrire une fonction `pascal` qui vérifie l'identité de Pascal : $\binom{n}{p} + \binom{n}{p+1} = \binom{n+1}{p+1}$ valable pour tous entiers `n, p ≥ 0`. Vérifie l'identité pour tous les entiers `n` et `p` entre 0 et 100.
- ④ Ecrire une fonction `maxBinomial(n)` qui renvoie la valeur maximale de tous les coefficients binomiaux $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots, \binom{n}{n}$
- ⑤ Ecrire une fonction `verifMaxBinomial(n)` qui vérifie que la valeur maximale de tous les coefficients binomiaux $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots, \binom{n}{n}$ est $\binom{n}{m}$ où `m` est le quotient la division entière de `n` par 2.

Minimum et maximum simultanés

Ecrire une fonction `min_max` qui prend en paramètre une liste `L` et retourne sous forme de liste la plus petite et la plus grande valeur de `L`.

Exemples :

```
1 [12, 81, 65, 9, 32] -> [9, 81]
2 [42] -> [42, 42]
3 [42, 42, 42, 42, 42] -> [42, 42]
```

Plus petit entier non nul (découpage en fonctions)

- ① Écrire une fonction `mini(L, i)` qui prend en paramètre

- une liste L d'entiers
- un indice valide positif i de la liste L

et renvoie le plus petit des éléments de la liste L et ayant un indice supérieur ou égal à i. Voici quelques exemples de comportement de la fonction :

```
1 [40, 10, 50, 20] 0 -> 10
2 [40, 10, 50, 20] 1 -> 10
3 [40, 10, 50, 20] 2 -> 20
4 [40, 10, 50, 20] 3 -> 20
```

- ② Écrire une fonction `plusPetitNonNul(L)` qui prend en paramètre une liste L d'entiers *non tous nuls* et qui renvoie le plus petit indice d'un élément non nul de L.

Voici quelques exemples de comportement de la fonction :

```
1 [50, 10, 0, 90] -> 0
2 [0, 0, 70, 0, 42] -> 2
3 [0, 0, 70] -> 2
4 [42] -> 0
```

- ③ En utilisant les deux fonctions précédentes, écrire une fonction `miniNonNul(L)` qui prend en paramètre une liste L d'entiers *non tous nuls* et qui renvoie le plus petit élément *non nul* de L.

Voici quelques exemples de comportement de la fonction :

```
1 [50, 0, 0, 90, 10] -> 10
2 [0, 80, 10, 60, 0] -> 10
3 [0, 0, 0, 40, 0, 30, 0] -> 30
4 [0, 0, 0, 40, 0, 50, 0] -> 40
5 [0, 0, 0, 40, 0, 0, 0] -> 40
6 [0, 0, 0, 0, 0, 0, 40] -> 40
7 [50, 80, 10, 60, 40] -> 10
8 [42] -> 42
```

Formule de Keith et Craver

La formule de Keith et Craver permet de déterminer le jour de la semaine (ie lundi, mardi, etc) correspondant à une date donnée (par exemple, le 14 juillet 1789 qui était un mardi).

Ci-dessous, en voici une implémentation sous forme de fonction Python. La fonction `kc` retourne sous forme d'entier (1 = lundi, 2 = mardi, ..., 7 = dimanche) le jour de la semaine correspondant à une date passée en paramètre comme suit : j (jour), m (mois) et a (année).

```
1 def kc(j, m, a):
2     z = a - (m<3)
3     return (j + 23*m//9 + 3 - 2*(m>=3) + a + z//4 - z//100 + z//400)%7 + 1
```

Voici un exemple d'utilisation :

```
1 def kc(j, m, a):
2     z = a - (m<3)
3     return (j + 23*m//9 + 3 - 2*(m>=3) + a + z//4 - z//100 + z//400)%7 + 1
4
5 # Test du jour de la semaine du 1er septembre 2016
6 print(kc(1, 9, 2016))
```

— Lignes 6 et 7 : le 1^{er} septembre 2016 est un jeudi (4^e jour de la semaine).

Cette fonction doit être utilisée telle quelle, sans chercher à comprendre comment elle fonctionne. Vous devez réécrire cette fonction pour pouvoir l'utiliser par la suite mais il est inapproprié de copier/coller le corps de la fonction lignes 2 et 3 dans votre propre code. Il est seulement attendu d'**utiliser** la fonction `kc`.

- ① Vérifier les dates suivantes :
 - vendredi 13 janvier 2016
 - mardi 14 juillet 1789
 - dimanche 10 mai 1981
 - jeudi 16 juillet 1998
 - lundi 1^{er} mai 2017
 - mardi 19 janvier 2038 (le bug de l'an 2038)
- ② En utilisant un appel à la fonction `kc`, écrire une fonction `est_vendredi13(m,a)` qui renvoie `True` si le 13 du mois `m` et de l'année `a` est un vendredi (et `False` sinon). Combien y-a-t-il de vendredis 13 dans l'année 2017 ?
- ③ Ecrire et tester une fonction `jour_semaine` qui accepte en argument une **liste date**, supposée de la forme `[jour, mois, année]` et qui retourne le jour de la semaine correspondant en toutes lettres (ex : *lundi, mardi, ..., dimanche*).

Par exemple, `jour_semaine(14, 7, 1789)` vaut `mardi`.

Calcul d'une durée

- ① Écrire une fonction `hms_to_s` qui retourne en secondes une durée exprimée en heures, minutes et secondes. Vérifier que `1h30m = 5400s` et `3h20m15s = 12015s`.
- ② Écrire une fonction `s_to_hms` qui prend en paramètre une durée exprimée en secondes et la retourne exprimée en heures, minutes et secondes sous forme de liste `[h, m, s]`. Vérifier que `s_to_hms(5400)` retourne la liste `[1, 30, 0]`, et que `s_to_hms(12015)` retourne la liste `[3, 20, 15]`.
- ③ Écrire une fonction `afficher_temps` capable d'afficher une liste `[h, m, s]` sous la forme `h heures m minutes et s secondes`. Par exemple, `afficher_temps([2, 42, 16])` affiche `2 heures 42 minutes et 16 secondes`.
- ④ Écrire un programme qui affiche, en une seule ligne de code, le temps écoulé entre 13h58m et 15h31m30s.

Liste croissante

Ecrire une fonction `est_croissante` qui détermine si une liste d'entiers passée en paramètre est croissante (i.e. renvoie `True`) et `False` sinon.

Qu'une liste `L` soit croissante signifie que $L[i] \leq L[i + 1]$ pour tout indice `i` pour lequel l'inégalité a un sens.

Exemples :

```
1 [5, 6, 6, 6, 10] -> True
2 [5, 6, 6, 6, 4] -> False
3 [42] -> True
```

Aucun multiple de 10

Écrire une fonction `aucunMultiple10` qui prend en paramètre une liste d'entiers et qui renvoie `True` si la liste ne contient aucun multiple de 10. Voici deux exemples de comportement de la fonction :

```
1 [42, 81, 33, 81] -> True
2 [42, 91, 75, 90, 33] -> False
```

Remplacer les nombres négatifs

Écrire une fonction `rempl_neg` qui prend en paramètre une liste d'entiers L , ne retourne rien mais remplace dans la liste L chaque entier négatif en son opposé, par exemple :

```
1 [-6, 2, 5, -3, 0, 2, 0, -1] -> [6, 2, 5, 3, 0, 2, 0, 1]
```

Ainsi, la fonction `rempl_neg` **modifie** la liste qu'elle reçoit en argument.

Un élément sur trois

Écrire une fonction f qui prend une liste L d'entiers et qui renvoie la liste M formée d'un élément sur 3 de la liste L en commençant par la fin. Exemples :

```
1 [19, 43, 25, 54, 71, 76, 88, 53, 67, 22, 90, 79, 12, 25] -> [25, 90, 53, 71, 43]
2 [19, 43, 25, 54] -> [54, 19]
3 [19] -> [19]
```

Liste sans répétition successive

Soit, par exemple, la liste L d'entiers :

[3, 7, 5, 5, 8, 8, 8, 8, 2, 5, 5, 5].

Elle contient des répétitions *successives* comme 8, 8, 8, 8 ou encore 5, 5, 5. La liste sans répétition successive est [3, 7, 5, 8, 2, 5].

Plus généralement, écrire une fonction $f(L)$, où L est une liste non vide d'entiers, qui renvoie la liste M formée des éléments de L sans ses répétitions de termes successifs. La liste M doit être une *nouvelle* liste et la liste L ne doit pas être modifiée par la création de M .

Exemples :

```
1 [7, 7, 5, 5, 8, 8, 8, 8, 5, 5, 5] -> [7, 5, 8, 5]
2 [7, 7, 7] -> [7]
3 [7, 0, 1] -> [7, 0, 1]
4 [7] -> [7]
```

ppcm

Écrire une fonction `ppcm` qui renvoie le plus petit commun multiple de deux entiers donnés. Par exemple, `ppcm(16, 60)` vaut 240.

Il est attendu d'écrire une version « naïve » (et non pas d'utiliser l'algorithme d'Euclide pour obtenir d'abord le pgcd). La version naïve consistera par exemple à examiner les multiples non nuls de a et à tester jusqu'à ce qu'un de ces multiples soit multiple de b .

Suite de Syracuse

① Soit la fonction f définie pour $n > 0$ entier par :

— $f(n)$ est la moitié de n si n est pair

— $f(n)$ vaut $3n + 1$ si n est impair

Par exemple $f(13) = 40$ ou encore $f(10) = 5$.

Coder en Python cette fonction. Tester avec $f(13)$ et $f(10)$.

- ② On itère la fonction f en partant d'un entier $n > 0$. Par exemple, si $n = 13$, on obtient :

1 13 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

La conjecture de Syracuse affirme que si on itère f depuis n'importe quel entier $n > 0$ alors, on tombera forcément sur 1 (ce qui se produit bien pour l'exemple ci-dessus).

Écrire une fonction `seq` qui

— prend n en paramètre,

— affiche les itérés de la suite depuis le premier itéré, à savoir n jusqu'au dernier, à savoir 1.

- ③ Écrire une fonction `saut(x)` qui renvoie le nombre d'itérations pour arriver à 1 quand la suite commence avec x (si $x = 13$, il y a donc 9 itérations pour arriver à 1).
- ④ Déterminer à l'aide de la fonction `saut(x)` le plus petit entier n tel qu'il faille exactement 100 itérations à partir de n pour arriver à 1 (vous devez trouver $n = 107$).

Le tri par sélection

Le *tri par sélection* est un algorithme de tri, utilisé par exemple pour trier une liste de nombres dans l'ordre croissant. Son principe est le suivant : on dispose d'une liste `t`, on en cherche le plus grand élément `m`, on le met « de côté », on recommence avec la liste restante en recherchant à nouveau son plus grand élément, puis on le place à gauche de `m`, et ainsi de suite jusqu'à ce que la liste initiale soit vide. La liste triée est alors la liste construite avec les maxima successifs.

Quand on programme un tri par sélection, pour éviter de multiples suppressions/recopies, au lieu de mettre « de côté » les maxima trouvés, on les déplace à l'intérieur même de la liste `t`. Plus précisément, soit une liste `t` de n entiers. Le tri par sélection consiste à :

— chercher le plus grand élément de `t`,

— l'échanger avec l'élément le plus à droite de `t`

— recommencer avec la liste formée des $n - 1$ premiers éléments de `t`.

- ① Écrire une fonction `maxi(t, p)` qui prend en paramètre une liste d'entiers `t` de longueur n , et un indice `p` valide de `t` (donc tel que $0 \leq p \leq n - 1$) et qui renvoie l'**indice** d'un élément maximum de la sous-liste de `t` qui s'étend de l'indice 0 (inclus) à l'indice `p` (inclus).
- ② (a) Écrire une procédure `echange(t, i, j)` qui prend en paramètre une liste d'entiers `t` et deux indices `i` et `j` d'éléments de `t` et qui échange les contenus dans `t` aux indices `i` et `j`.
- (b) Écrire le code d'un tri par sélection avec une procédure `selection(t)` et qui utilise `maxi` et `echange`.
- (c) Générer une liste `L` de 30000 entiers et lui appliquer la fonction `selection`. Que pensez-vous de la vitesse d'exécution ?

Tirage du loto

Écrire une fonction `tirerLoto` qui génère un tirage aléatoire des 6 numéros d'un loto.

Rappel : si `L` est une liste d'entiers et `x` un entier alors l'expression `x in L` vaut `True` si l'entier `x` est dans la liste `L` et `False` sinon.

PILE quatre fois de suite

- ① Ecrire une fonction `lancer_piece` qui simule le lancer d'une pièce de monnaie équilibrée. La fonction retournera la chaîne "PILE" ou la chaîne "FACE".
- ② On dispose d'une pièce de monnaie. On cherche à savoir quel est, en moyenne, le minimum de lancers qu'il faut effectuer pour obtenir 4 fois *FACE* **consécutivement**.
Créer une fonction `attente` qui effectue une suite de lancers de la pièce et qui renvoie le nombre de lancers qu'il faut faire pour obtenir pour la première fois 4 fois FACE de suite.
- ③ Ecrire une fonction `test` qui itère n l'expérience de la fonction `attente` la fonction `attente`. Vérifier qu'il faut en moyenne 30 lancers pour obtenir 4 fois de suite FACE.

Tester sur $n = 10^5$ expériences.

Jeu du 421

Le jeu du 421 consiste à obtenir avec 3 dés une combinaison contenant une fois le chiffre 4, une fois le chiffre 2 et une fois le chiffre 1. Par exemple, si vous jetez trois dés simultanément et que vous obtenez la combinaison 2, 1 et 4, vous avez gagné ; si vous obtenez 1, 4, 1 alors votre coup n'est pas gagnant.

L'exercice va demander combien de jets de 3 dés sont nécessaires, en moyenne, pour faire un 421.

- ① Ecrire une fonction `est421` qui prend en paramètre une liste de trois entiers et renvoie
 - `True` si la liste, à la fois, contient 4, contient 2 et contient 1
 - `False` sinon.

Ainsi :

```
— est421([6, 2, 1]) = False
— est421([1, 1, 1]) = False
— est421([2, 4, 1]) = True
— est421([1, 2, 4]) = True
— est421([4, 2, 1]) = True
```

- ② Ecrire une fonction `tirage421` qui renvoie, sous forme de liste, un tirage aléatoire de trois dés. Par exemple, si vous tirez les nombres

```
1 6, 2, 2
```

alors la fonction doit renvoyer la liste `[6, 2, 2]`. Les nombres peuvent avoir été tirés simultanément ou l'un après l'autre, c'est indifférent.

- ③ Ci-dessous, on appelle *suite gagnante* toute succession de lancers de trois dés, autant de fois que nécessaire, pour « sortir » 421. Par exemple, la suite de 7 lancers suivants est une suite gagnante :

```
1 523 144 643 235 451 252 214
```

Écrire une fonction `nombreSuitesGagnantes` qui ne prend aucun paramètre et qui

- simule une suite gagnante,
- renvoie le nombre de lancers de la suite gagnante

Par exemple, `nombreSuitesGagnantes` renvoie 7 dans le cas de la suite gagnante ci-dessus.

- ④ On fait 1000 suites gagnantes. Calculer le nombre moyen de lancers d'une suite gagnante (le résultat théorique est 36).

Liste monotone

Une liste d'entiers est monotone ou bien si elle est croissante ou bien décroissante.

Ecrire une fonction `est_monotone` qui détermine si une liste d'entiers est monotone (i.e. renvoie `True` et `False` sinon).

Il pourra être utile de remarquer que deux nombres ont des signes différents si leur produit est strictement négatif.

Exemples :

```
1 [5, 5, 6, 6, 6, 10] -> True
2 [5, 5, 6, 6, 6, 6] -> True
3 [5, 5, 6, 6, 6, 5] -> False
4 [42, 5, 6, 6, 6, 5] -> False
5 [42] -> True
6 [42, 42, 42, 42, 42] -> True
7 [5, 4] -> True
8 [4, 5] -> True
9 [5, 4, 3, 3, 0] -> True
10 [5, 4, 3, 42, 43, 44] -> False
11 [5, 4, 3, 3, 42] -> False
12 [] -> True
```

Supprimer les doublons

Ecrire une fonction `eliminer_doublons(L)` qui partant d'une liste d'entiers retourne une liste composée des valeurs de `L`, mais dans laquelle les multiples valeurs égales de `L` (s'il y en a) n'ont été copiées qu'une seule fois. On dira ainsi de la liste retournée qu'elle ne contient pas de doublon. Exemples :

```
1 [42, 81, 42, 65, 12, 81, 31, 42] -> [42, 81, 65, 12, 31]
2 [42] -> [42]
3 [42, 42, 42, 42, 42] -> [42]
```

Fractions

Dans cet exercice, une fraction sera représentée par une liste de 2 entiers positifs :

[numérateur, dénominateur].

Par exemple, la fraction $\frac{22}{7}$ sera représentée par la liste `[22, 7]`. Plus généralement, la fraction $\frac{a}{b}$ sera représentée par la liste `[a, b]`.

- ① Écrire une fonction `aff` qui affiche une fraction `frac` donnée en paramètre. Par exemple, `aff([22,7])` affichera `22 / 7`.
- ② Écrire une fonction `add(frac1, frac2)` qui retourne la somme des fractions `frac1` et `frac2`. On appliquera la formule suivante : $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$. Par exemple, `add([1,3], [7,10])` retournera `[31,30]`. On ne cherchera pas à obtenir une fraction simplifiée (autrement dit une fraction irréductible).
- ③ Écrire une fonction `harmonique(n)` qui calcule la somme $1 + \frac{1}{2} + \dots + \frac{1}{n}$. Vérifier que `harmonique(6)` affiche `1764 / 720`.
- ④ Écrire une fonction `pgcd(a, b)` qui retourne le plus grand diviseur commun de deux entiers `a` et `b`. Par exemple, `pgcd(140, 100) = 20`. On procédera de la manière naïve suivante : parcourir tous les entiers `d` entre 1 et, par exemple `a`, et stocker `d` dans une variable chaque fois que, simultanément, `a` est multiple de `d` et que `b` est multiple de `d`.

- ⑤ Écrire une fonction `simplifier(frac)` qui simplifie la fraction `frac` reçue en paramètre. Par exemple, `simplifier([140, 100])` doit renvoyer `[7,5]`.

Si vous essayez de simplifier `harmonique(12)`, vous allez constater que l'exécution va être extrêmement longue.

Quitte à récrire légèrement le code de la fonction `add`, parvenir à simplifier `harmonique(12)`

Vendredi 13

On se propose d'écrire un code qui recherche les 10 prochains vendredis 13.

On codera chaque mois par un entier entre 1 et 12, chaque jour par un entier entre 1 et 31 et chaque jour de la semaine par un code (lundi = 1, mardi = 2, ..., dimanche=7).

- ① Ecrire une fonction `lendemain(js, j, m, a)` qui prend une date en paramètres et renvoie une liste définissant la date du lendemain. Par exemple `lendemain(2, 8, 12, 2015)` demande la date du lendemain du mardi 8 décembre 2015 et doit renvoyer la liste `[3, 9, 12, 2015]` car le lendemain est le mercredi 9 décembre 2015.

Les paramètres ont les significations suivantes :

- le jour de la semaine `js` (un entier entre 1 et 7)
- le numéro `j` du jour dans le mois (un entier entre 1 et 31)
- le numéro `m` du mois (un entier entre 1 et 12)
- l'année `a`.

La fonction traitera successivement les cas suivants :

- le 31 décembre,
- la fin des mois autres que décembre et ayant 30 ou 31 jours,
- la fin du mois de février,
- tous les autres jours.

On pourra utiliser les listes suivantes :

- `mois30 = [4, 6, 9, 11]`
- `mois31 = [1, 3, 5, 7, 8, 10]`

sachant qu'on peut tester l'appartenance d'un objet `x` dans une liste `L` par :

`x in L`

On rappelle qu'une année `a` est bissextile si le booléen

```
1 a % 400 == 0 or (a % 4 == 0 and a % 100 != 0)
est True.
```

Pour trouver le jour de la semaine du lendemain, on pourra utiliser un reste de division par 7 mais ce n'est pas indispensable, on peut toujours faire une instruction `if`.

On effectuera les tests suivants :

```
1 (2, 8, 12, 2015) -> [3, 9, 12, 2015]
2 (5, 1, 1, 2016) -> [6, 2, 1, 2016]
3 (4, 31, 12, 2015) -> [5, 1, 1, 2016]
4 (7, 31, 1, 2016) -> [1, 1, 2, 2016]
5 (6, 30, 1, 2016) -> [7, 31, 1, 2016]
6 (1, 29, 2, 2016) -> [2, 1, 3, 2016]
7 (7, 28, 2, 2016) -> [1, 29, 2, 2016]
8 (6, 28, 2, 2015) -> [7, 1, 3, 2015]
9 (6, 16, 7, 2016) -> [7, 17, 7, 2016]
```

- ② Déterminer les 10 prochains vendredis 13 à partir d'aujourd'hui. Vérifiez avec un calendrier.

6 Récursivité

a) Introduction à la récursivité

To iterate is human, to recurse divine. (L. Peter Deutsch)

Il est courant qu'une fonction f , lors de son exécution, appelle une fonction, disons g . Et il est possible que la fonction g soit f elle-même. Une telle fonction s'appelant elle-même est dite *récursive*.

Voici un exemple artificiel de fonction récursive.

Soit la fonction f telle que $f(n) = 1 + 2 + \dots + n$, somme des entiers entre 1 et n inclus. Par exemple,

- $f(3) = 1 + 2 + 3 = 6$,
- $f(4) = 1 + 2 + 3 + 4 = 10$,
- $f(10) = 1 + 2 + 3 + \dots + 9 + 10 = 55$.

On va implémenter en Python la fonction f dans une version *récursive*. La construction algorithmique de f est basée sur l'observation suivante : pour connaître, par exemple, la somme S des entiers de 1 à 4, il suffit de connaître la somme T des entiers de 1 à 3 et dans ce cas $S = T + 4$. Autrement dit

$$f(4) = f(3) + 4$$

ce qui n'est qu'un cas particulier de la relation suivante, valable pour $n \geq 2$:

$$f(n) = f(n - 1) + n$$

Cette relation est appelée parfois *relation de récurrence*. Il est essentiel de noter que la relation précédente ne s'applique pas si $n = 1$ puisque $f(0)$ n'a pas été définie. Toutefois, $f(1)$ existe et vaut 1.

Ci-dessous, voici un code qui implémente la fonction f en Python en utilisant la relation ci-dessus :

```
1 def f(n):
2     if n == 1:
3         return 1
4     else:
5         return n+f(n-1)
6
7 print(f(3))
8 print(f(4))
9 print(f(10))
```

```
10 6
11 10
12 55
```

- ligne 5 : la fonction f s'appelle elle-même.

On notera que l'implémentation de `f` suit exactement la relation de récurrence ci-dessus (cf. ligne 5 du code), y compris pour son cas d'exclusion (`n = 1`, cf. lignes 2-3).

Analyse de l'exécution de `f`

Pour comprendre comment fonctionne la récursivité, on va examiner comment est calculé `f(3)` : Les appels de la fonction `f` s'empilent :

- Le calcul de `f(3)` est lancé. On a `n = 3`. Comme `n != 1`, `f` renvoie `3 + f(2)`. Donc `f(2)` est appelé.
- Le calcul de `f(2)` est lancé. On a `n=2`. Comme `n != 1`, `f` renvoie `2 + f(1)`. Donc `f(1)` est appelé.
- Le calcul de `f(1)` est lancé. On a `n = 1`. Par suite, `f` renvoie 1.

A ce stade, les appels `f(2)` et `f(3)` sont en suspens, ie ils n'ont pas renvoyé leur résultat. Seul l'appel `f(1)` a renvoyé son résultat. Ensuite, les appels se dépilent :

- donc `f(2)` renvoie `2 + f(1) = 2 + 1 = 3`.
- donc `f(3)` renvoie `3 + f(2) = 3 + 3 = 6`. Le premier appel lancé dans le code (ici `f(3)`, cf. ligne 7) est cette fois terminé et donc l'exécution s'arrête.

Pour en revenir au calcul de `f(3)`, la pile d'appels contient 3 appels lorsque `f(1)` est appelé. Ensuite, la pile d'appels perd un appel à chaque fois que le **return** de la ligne 5 est exécuté. On rappelle qu'un **return** a pour effet d'interrompre l'exécution d'une fonction.

L'appel récursif figure à la ligne 5. La récursion finit par s'arrêter à cause de la condition à la ligne 3 où `f` n'est pas rappelée.

b) Outil de visualisation des appels récursifs

Le site [Pythontutor](#) propose un outil en ligne permettant de visualiser l'exécution de votre code et l'état de la mémoire pendant l'exécution, en particulier de conteneurs (listes, chaînes, etc). Cet outil est en particulier très pratique pour pouvoir observer la pile des appels d'une fonction récursive.

Pour utiliser l'outil :

- se rendre sur cette [page](#)
- coller votre code dans la zone de texte
- cliquer sur le bouton *Visualize Execution*

Apparaît alors une interface :

!

qui permet de progresser pas à pas dans l'exécution du code. Au fur et à mesure que la récursion se déroule, le pile des appels augmente ou diminue.

c) Ecrire un algorithme récursif

Pour écrire un algorithme récursif résolvant un problème `X` appliqué à un objet `N`, on identifie les deux éléments suivants :

- le *sous-problème* : on identifie le même problème que le problème `X` mais appliqué un objet `M` de « taille » inférieure et dont la *résolution* permet de résoudre le problème `X` appliqué à l'objet `N`

- le *cas de base* : on isole le cas du problème X appliqué à un objet de taille telle que le problème pour X ne peut être ramené à un sous problème ; il s'agit en quelque sorte d'un *cas irréductible*.

Par exemple, dans le problème d'introduction, le problème X est de calculer la somme S des n premiers entiers et l'objet N est l'entier positif ou nul n ; on se rend compte que pour connaître S , il suffit de savoir résoudre le même problème mais pour un objet plus petit, à savoir $n - 1$ puisque si l'on connaît la somme T des $n - 1$ premiers entiers alors $S = T + n$. Cependant, la règle précédente ne s'applique que si on peut effectivement décomposer le problème ce qui suppose que n vaut au moins 1 et ce qui fournit le cas de base.

d) Résolution de problème par des algorithmes récursifs

Certains problèmes sont définis de manière éminemment récursive ; voici quelques exemples :

- le tri rapide (*quicksort*)
- le tri fusion (*mergesort*)
- le problème des tours de Hanoï
- la recherche dichotomique dans une liste ordonnée
- l'évaluation d'expressions arithmétiques
- tracé de certains fractals (courbes de Von Koch, éponge de Sierpinsky, etc)
- le parcours en profondeur dans un graphe
- l'algorithme de Karatsuba de multiplication de deux grands entiers
- etc.

Pour certains problèmes, qu'ils soient définis de manière récursive ou pas, l'algorithme de résolution peut être implémenté en version récursive ou en version itérative. C'est par exemple le cas :

- d'une recherche dichotomique,
- d'un parcours en profondeur,
- d'une conversion d'un entier en base 2,
- du calcul du pgcd par l'algorithme d'Euclide.
- l'algorithme de transformée de Fourier rapide et discrète
- etc.

Le choix d'une version de l'algorithme plutôt que l'autre sera dicté par les facteurs suivants :

- facilité de codage,
- performance.

Dans certains cas, un algorithme récursif sera beaucoup plus concis que son équivalent itératif. Mais concis ne signifie pas forcément plus lisible ou plus intelligible.

On prendra garde qu'en Python dans son implémentation la plus répandue la pile est de taille limitée et, qu'un débordement de la pile peut survenir. Par ailleurs, même en augmentant la taille de la pile, une implémentation récursive peut avoir des performances très dégradées par rapport à son équivalent itératif.

Enfin, itération et récursion ne sont pas antinomiques. Par exemple, de nombreux problèmes de combinatoire et de dénombrement utilisent des appels récursifs dans des structures itératives.

e) Récursivité inefficace

Le problème de la suite de Fibonacci est un cas typique où une implémentation récursive conduit à des performances extrêmement dégradées.

La suite de Fibonacci est la suite d'entiers dont les premiers termes sont les suivants

1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...

Les deux premiers termes de la suite sont 1 et 1 et ensuite chaque terme de la suite est la somme des deux précédents. Ainsi, ci-dessus, on lit que $144 = 55 + 89$.

Le problème est d'écrire une fonction qui renvoie le n -ème terme de la suite de Fibonacci. Par exemple, si $n = 8$, le programme doit renvoyer 21.

Une implémentation récursive apparaît immédiatement :

- sous-problème : il suffit de connaître les deux termes précédents pour connaître le terme courant donc si $n \geq 3$, le problème se décompose en deux sous-problèmes de « taille » inférieure ;
- cas de base : les cas $n = 1$ et $n = 2$ doivent être traités directement.

D'où le code suivant :

```
1 def fib(n):
2     if n <= 2:
3         return 1
4     return fib(n-1) + fib(n-2)
5
6 for n in range(1,15):
7     print(fib(n), end=' ')
8 print()
```

```
9 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- Ligne 2 : les cas de base $n = 1$ et $n = 2$.
- Ligne 4 : la décomposition en sous-problèmes (cas $n - 1$ et $n - 2$) et la résolution.
- Ligne 9 : aucun problème apparent de performance (pas d'attente d'affichage du résultat).

En réalité, le programme ci-dessus est extrêmement peu efficace car il répète un grand nombre de fois les mêmes appels. Par exemple, le calcul de $f(5)$ engendre le calcul séparé de $f(4)$ puis de $f(3)$ mais le calcul de $f(4)$ va entraîner le calcul de $f(3)$ qui sera donc *recalculé* ultérieurement. En modifiant le programme ci-dessus, il est même possible de calculer le nombre total d'appels effectués pour calculer $\text{fib}(n)$:

```
1 N=0
2
3 def f(n):
4     global N
5     N+=1
6     if n <= 2:
7         return 1
8     else:
9         return f(n-1)+f(n-2)
10
11 print(f(32))
12 print("Nombre d'appels :", N)
```

```
13 2178309
14 Nombre d'appels : 4356617
```

Ainsi, pour calculer le 32^e terme de la suite de Fibonacci, il faut appeler la fonction `fib` plus de 4 millions de fois, ce qui entraîne des performances de calculs extrêmement dégradées. Noter qu'il ne s'agit pas d'un problème de taille de la pile dont la hauteur va rester au maximum d'une trentaine d'éléments.

Bien qu'il existe une alternative encore récursive corrigeant le problème ci-dessus, dans le cas présent, la meilleure façon de coder la suite de Fibonacci est encore d'utiliser l'implémentation itérative :

```
1 def fibo(n):
2     if n==1 or n==2:
3         return 1
4     x=y=1
5     for i in range(0,n-2):
6         z=x+y
7         x=y
8         y=z
9     return z
10
11 print(fibo(32))
12 2178309
```

f) Récursivité illustrée par la conversion en base b

Soit à déterminer la liste des chiffres de la représentation en base b d'un entier x . Par exemple, si $x = 13$ et $b = 2$ alors les chiffres de la représentation de x en base b sont les éléments de la liste $[1, 0, 1, 1]$ puisque $13 = 1 + 0 + 4 + 8$. On conviendra que la liste commencera par le chiffre des unités.

L'algorithme de conversion consiste à faire des divisions successives par b , en commençant par x et à arrêter les divisions juste avant que le quotient ne vaille 0. Les chiffres de la représentation en base b de x sont alors les restes successifs.

Ainsi

- si $x \geq b$ alors les chiffres de x en base b sont le reste de la division de x par b suivi des chiffres du quotient de x par b ;
- si $0 \leq x < b$ alors x admet un seul chiffre en base b , à savoir x lui-même.

Ceci fournit un algorithme récursif de conversion en base b :


```

1 def en_base_b(b, n):
2     if n<b:
3         return [n]
4     return [n%b] + en_base_b(b, n//b)
5
6
7 print("-----")
8 b=10
9 print("base", b, ":")
10 print()
11 for n in [42, 2038]:
12     print(n, "->", en_base_b(b, n))
13
14 print("-----")
15 b=2
16 print("base", b, ":")
17 print()
18 for n in [0, 1, 13, 42, 63, 64]:
19     print(n, "->", en_base_b(2, n))
20
21 -----
22 base 10 :
23
24 42 -> [2, 4]
25 2038 -> [8, 3, 0, 2]
26 -----
27 base 2 :
28
29 0 -> [0]
30 1 -> [1]
31 13 -> [1, 0, 1, 1]
32 42 -> [0, 1, 0, 1, 0, 1]
33 63 -> [1, 1, 1, 1, 1, 1]
34 64 -> [0, 0, 0, 0, 0, 0, 1]

```

- On traite d'abord du cas le plus abordable, la base 10, cf. lignes 8-12. Ainsi, cf. ligne 24, on obtient bien la liste des chiffres en base 10 de l'entier 2038.
- Lignes 1-4 : la fonction `en_base_b` renvoie la liste de chiffres de la conversion de `n` en base `b`. Le premier chiffre qui apparaît dans la liste (cf. `[n%b]`) est le chiffre des unités ; par exemple, ligne 23, le chiffre des unités de 42 en base 10 est bien 2.
- Lignes 2-3 : on traite d'abord le cas où le quotient de la division de `n` par `b` serait nul : dans ce cas, il n'y a qu'un chiffre, que l'on renvoie dans une liste (ligne 3).
- Ligne 4 : le cas général consiste à calculer le chiffre des unités de `n` qui est `n%b`, à le placer seul dans une liste (cf. `[n%b]` à la ligne 1), à déterminer récursivement la liste des chiffres suivants (ce sont ceux du quotient de `n` par `b`) et à réunir les listes avec l'opérateur `+` sur les listes.
- Lignes 14-19 : on choisira des exemples de tests permettant d'assurer une vérification immédiate (lignes 28, 29, 32 et 33). Par exemple, ligne 33, comme 64 est une puissance de 2, tous ses chiffres sont nuls sauf le chiffre dominant.

g) Récursivité illustrée par la recherche dichotomique

Etant donné une liste L d'objets **triés par ordre croissant** et un objet x , l'algorithme de recherche dichotomique (*binary search* en anglais) est un algorithme efficace de recherche de la présence x dans la liste L .

Etude d'un exemple

Montrons sur un exemple le fonctionnement d'une recherche dichotomique. Le principe de base est qu'à chaque étape de l'algorithme, on va choisir la « bonne moitié » de la liste, celle où se trouve x .

Soit la liste croissante $L = [12, 31, 46, 53, 81, 81, 82]$ et soit $x = 42$. On découpe la liste en les deux listes suivantes qui sont moitié moins grandes que la liste L :

$$[12, 31, 46] \text{ et } [53, 81, 81, 82]$$

En comparant avec 46 (dernier terme de la première liste), on voit que x est forcément dans la première liste, disons $M = [12, 31, 46]$.

On découpe à nouveau cette liste en les deux listes suivantes qui sont moitié moins grandes que la liste M :

$$[12] \text{ et } [31, 46]$$

En comparant avec 12 (dernier terme de la première liste), on voit que x est forcément dans la deuxième liste $[31, 46]$.

On découpe à nouveau cette liste en deux :

$$[31] \text{ et } [46]$$

En comparant avec 31 (dernier terme de la première liste), on voit que x est forcément dans la deuxième liste $[46]$. Comme cette liste est de taille 1, il suffit de regarder si l'élément $x = 42$ est cet élément : ce n'est pas le cas, donc la liste initiale ne contient pas l'élément 42.

Principe de la recherche dichotomique

Le principe est le suivant : on découpe la liste en deux sous-listes de taille « environ » la moitié de la liste initiale (le « environ » est précisé ci-dessous) et on identifie la sous-liste M susceptible de contenir x et on recommence la même recherche dans cette nouvelle sous-liste.

Pour préciser le « environ » ci-dessus, si n est la taille de L le découpage par moitiés se fera suivant les longueurs définies par les décompositions ci-dessous :

- si $n = 2k$ est pair alors on décomposera en $n = k + k$
- si $n = 2k + 1$ est pair alors on décomposera en $n = k + (k + 1)$

et en particulier, on décide, conventionnellement, que la liste de gauche n'est jamais plus grande que celle de droite.

Noter que si les indices de L commencent à 0 alors, avec les notations $n = 2k$ ou $n = 2k + 1$ ci-dessus, la longueur de la première sous-liste M du découpage est toujours k et donc l'indice i du dernier élément de M est $k-1$. Or, que n soit pair ou pas, k est le quotient de la division entière de n par 2 et donc, $i = n//2 - 1$ où $//$ est la division entière.

Implémentation de la recherche dichotomique

L'étude ci-dessus de la recherche dichotomique d'un élément x dans une liste croissante L conduit au code suivant :

```
1 def dichotomie(L, debut, n, x):
2     if n==1:
3         return L[debut] == x
4     k= n//2
5     if x <= L[debut + k-1]:
6         return dichotomie(L, debut, k, x)
7     else:
8         return dichotomie(L, debut+k, n-k, x)
9
10 L=[1, 3, 4, 6, 6, 6, 9, 10]
11
12 for i in range(0,12):
13     print(i, dichotomie(L, 0, len(L), i))
14 0 False
15 1 True
16 2 False
17 3 True
18 4 True
19 5 False
20 6 True
21 7 False
22 8 False
23 9 True
24 10 True
25 11 False
```

- Lignes 1 à 8 : `dichotomie` renvoie `True` si x est présent dans la sous-liste de la liste L commençant à l'indice `debut` de L et ayant n éléments. L est la liste initiale et x est l'élément cherché; sinon, `dichotomie` renvoie `False`
- La fonction `dichotomie` est récursive : elle s'appelle sur l'une des sous-listes (cf. lignes 6 et 8)
- Lignes 2-3 : le cas de base : si la liste est de taille 1, il n'y a pas de sous-liste et, par simple examen (ligne 3) il est possible de répondre.
- Ligne 4 : il faut introduire cette variable k pour éviter le recalcul aux lignes suivantes.
- Lignes 5-6 : la fonction examine si x est susceptible de se trouver dans la liste gauche (ligne 5) et si c'est possible, la fonction `dichotomie` est rappelée pour chercher x dans la sous-liste gauche
- Ligne 8 : si x ne peut être dans la sous-liste gauche, il ne peut être que dans la sous-liste droite d'où un nouvel appel de la fonction `dichotomie` pour chercher x dans la sous-liste droite.

Comme la taille de la liste diminue strictement à chaque appel, elle finit par valoir 1.

Complexité de la recherche dichotomique

On peut démontrer que si la liste initiale est de taille n alors le nombre d'appels effectués par la fonction `dichotomie` est environ $\log_2(n)$, le logarithme en base 2 de n . Autrement dit, si

n est proche de 2^p alors p appels sont effectués. Par exemple, une liste de 1 million d'entiers (environ 2^{20}) nécessite une vingtaine d'appels au plus. Voilà en quoi l'algorithme de la recherche dichotomique est qualifié d'efficace, en particulier en comparaison de la recherche séquentielle naïve consistant à parcourir toute la liste depuis son début et jusqu'au point nécessaire, dans le cas d'une liste d'un millions d'entrées, cela peut nécessiter jusqu'à un million de comparaisons au lieu d'un vingtaine pour la recherche dichotomique.

h) Récursivité illustrée par le produit cartésien

Soit à composer un repas constitué d'une seule entrée, d'un seul plat et d'un seul dessert. Les choix sont les suivants :

Entrée	Plat	Dessert
salade	poulet	yaourt
taboulé	saumon	orange
	omelette	mousse

Quelle est la liste de tous les repas possibles ? Il suffit de choisir une entrée parmi 2, puis pour chaque choix précédent, un plat principal parmi 3 (ce qui fait 6 choix possibles au total) et pour chacun de ces choix, un des 3 desserts possibles ce qui fait au total $6 \times 3 = 18$ menus possibles.

Il s'agit mathématiquement de construire le produit cartésien $E \times P \times D$ des trois ensembles $E = \{\text{salade, taboulé}\}$, $P = \{\text{poulet, saumon, omelette}\}$ et $D = \{\text{yaourt, orange, mousse}\}$, autrement dit l'ensemble des triplets de la forme (entrée, plat, dessert).

Plus généralement, on se donne p listes d'objets L_0, \dots, L_{p-1} et on cherche à construire la liste M formée de toutes les listes possibles de la forme $[x_0, x_1, \dots, x_{p-1}]$ où chaque x_i est un élément de L_i . Une telle liste de p éléments sera appelée un p -uplet.

Bien sûr, il est possible de donner un code naïf répondant à la question initiale où $p = 3$:

```

1 entrees = ["salade", "taboulé"]
2 plats = ["poulet", "saumon", "omelette"]
3 desserts = ["yaourt", "orange", "mousse"]
4
5 for e in entrees:
6     for p in plats:
7         for d in desserts:
8             print(e, p, d)

```

qui affiche

```

1  salade poulet yaourt
2  salade poulet orange
3  salade poulet mousse
4  salade saumon yaourt
5  salade saumon orange
6  salade saumon mousse
7  salade omelette yaourt
8  salade omelette orange
9  salade omelette mousse
10 taboulé poulet yaourt
11 taboulé poulet orange
12 taboulé poulet mousse
13 taboulé saumon yaourt
14 taboulé saumon orange
15 taboulé saumon mousse
16 taboulé omelette yaourt
17 taboulé omelette orange
18 taboulé omelette mousse

```

Mais ce code n'est pas transposable pour un p quelconque.

Supposons que les p listes d'objets L_0, \dots, L_{p-1} soient données sous forme d'une liste $L = [L_0, \dots, L_{p-1}]$.

Il existe un algorithme récursif répondant à la question. En effet, la liste de tous les p -uplets possibles s'obtient en disposant de tous les $p - 1$ uplets possibles auxquels on adjoint un élément quelconque du dernier ensemble. Par exemple, si j'ai l'ensemble de tous les menus où l'entrée et le plat principal sont donnés, il suffit de rajouter un quelconque dessert aux menus trouvés pour obtenir tous les menus complets.

Pour le cas de base qui correspond à $p = 1$ alors la liste cherchée n'est autre que $[L[0]]$ où $L[0]$ est l'unique liste.

D'où le code (commenté) suivant :

```

1 def f(L):
2     p=len(L) # Le nombre de listes
3     if p==1:
4         # S'il y a une seule liste
5         # on renvoie une liste composée
6         # d'une copie de la liste
7         # -----
8         # L'unique liste
9         L0=L[0]
10        r=[]
11        # On copie dans r
12        for i in range(len(L0)):
13            # observer les crochets de liste indispensables
14            r.append([L0[i]])
15        return r
16    LL=[] # Préparation de l'appel récursif
17    # -----
18    # on extrait les p-1 premières listes
19    for i in range(p-1):
20        LL.append(L[i])
21    # appel récursif : M est
22    # la liste des éléments du produit cartésien des p-1 ensembles
23    M=f(LL)
24    r=[] # le futur résultat attendu
25    for X in M:
26        for a in L[p-1]:
27            # X+ [a] est e (p-1)-uplet complété par a
28            # noter les crochets de liste
29            r.append(X+[a])
30    return r
31
32
33 entrees = ["salade", "taboulé"]
34 plats =["poulet", "saumon", "omelette"]
35 desserts = ["yaourt","orange", "mousse"]
36
37 L=[entrees, plats, desserts]
38 repas = f(L)
39
40 for e,p,d in repas:
41     print(e,p,d)

```

qui affiche

```

1 salade poulet yaourt
2 salade poulet orange
3 salade poulet mousse
4 salade saumon yaourt
5 salade saumon orange
6 salade saumon mousse
7 salade omelette yaourt
8 salade omelette orange
9 salade omelette mousse
10 taboulé poulet yaourt
11 taboulé poulet orange
12 taboulé poulet mousse
13 taboulé saumon yaourt
14 taboulé saumon orange
15 taboulé saumon mousse
16 taboulé omelette yaourt
17 taboulé omelette orange
18 taboulé omelette mousse

```

i) Limitation du nombre d'appels récursifs

Il existe de nombreuses situations algorithmiques où il est efficace d'utiliser une fonction récursive. Toutefois, en Python, la pile d'appels ne doit pas dépasser une certaine limite qui, par défaut, est de 1000. Sinon, on obtient une erreur :

```

1 def f(n):
2     # Calcule 1+2+...+n
3     if n==0:
4         return 0
5     else:
6         return n+f(n-1)
7
8 print(f(1200))

```

qui affiche

```

1 RuntimeError: maximum recursion depth exceeded in comparison

```

Ce problème est dû à un débordement de la pile des appels⁵ et se retrouve, à des degrés divers, dans de nombreux langages de programmation non fonctionnels (C/C++, Java, Javascript, etc).

EXERCICES TYPE

Dis « Bonjour ! »

Ecrire une fonction récursive `bonjour(n)` qui affiche `n` fois le message **Bonjour !**.

Solution

5. Le fameux *stackoverflow* mais qui peut aussi se manifester sans appels récursifs !

Afficher n fois de suite **Bonjour !** revient à l'afficher une fois puis à redemander l'affichage $n - 1$ fois (ce qui est le sous-problème). Cette règle est valable pour $n \geq 1$ (en effet, si $n = 1$ alors afficher $n - 1$ fois n'aura aucun effet). On pourrait penser qu'il y a un cas de base mais en fait ce n'est pas vraiment le cas car afficher 0 fois est équivalent à ce que la fonction n'ait aucune action. D'où le code suivant :

```
1 def bonjour(n):
2     if n>0:
3         print("Bonjour !")
4         return bonjour(n-1)
5
6 bonjour(1)
```

On vérifie que si $n = 0$, il n'y a aucun affichage comme attendu.

Maximum en récursif

Ecrire une fonction récursive `maxi(L, p)` qui calcule le plus grand des éléments d'indices $i \geq p$ d'une liste L non vide d'entiers et où $p \geq 0$ est censé être un indice valide de L . Appliquer au calcul du plus grand élément d'une liste.

Solution

L'idée est que si la liste, entre l'indice p et la fin de la liste, contient au moins deux éléments, alors le maximum de ces éléments s'obtient en prenant le plus grand des éléments parmi le deux éléments suivants :

- l'élément a à l'indice p
- l'élément m qui est le plus grand des éléments de L d'indices à partir de p .

Or justement, l'élément m qui est un plus grand élément peut se calculer à l'aide de la fonction `maxi` mais sur une quantité plus petite de nombres.

Le cas de base est le cas qui n'est pas concerné par la situation précédente, c'est-à-dire le cas où p est le dernier indice de la liste (il n'y a rien derrière). Dans ce cas, `maxi(L, p)` va renvoyer ce dernier élément puisque c'est le seul élément à comparer.

D'où le code ci-dessous :


```

1 def maxi(L,p):
2     n=len(L)
3     if p==n-1:
4         return L[n-1]
5     else:
6         a=L[p]
7         m=maxi(L,p+1)
8         if m>a:
9             return m
10        else:
11            return a
12
13
14 tests=[[5, 7, 3], [7, 5, 3], [5,3, 7], [7]]
15
16 for L in tests:
17     print(L, '->', maxi(L, 0))

```

On notera l'exécution du **else** suppose que la liste contient au moins deux éléments (rappelons que **p** est censé être un indice valide).

Nombre de rangements « confortables »

Observez d'abord cet exemple : soit une suite de 7 cases vides, notée comme ceci

[0, 0, 0, 0, 0, 0, 0]

On veut déterminer le nombre de façons différentes de ranger 3 boules dans les cases en sorte que deux boules ne soient jamais dans des cases côte-à-côte ; un tel rangement sera dit *confortable*. Une boule étant représentée par un 1, le rangement

[0, 1, 0, 1, 0, 1, 0]

ou encore le rangement

[1, 0, 0, 0, 1, 0, 1]

sont donc confortables ; en revanche, le rangement

[1, 0, 0, 1, 1, 0, 0]

ne l'est pas car la 4^e et la 5^e cases sont des cases voisines et occupées par des boules.

Plus généralement, on cherche à déterminer le nombre de rangements *confortables* de **k** boules dans une succession de **n** cases vides, autrement dit, les rangements où deux boules ne sont jamais placées dans des cases immédiatement voisines.

Résoudre ce problème en définissant une fonction récursive **f(n, k)** qui renvoie le nombre de rangements *confortables*. On raisonnera en considérant deux cas :

- ou bien la dernière case est vide et il suffit de placer les k boules dans les $n - 1$ premiers emplacements
- ou bien la dernière case est occupée par une boule et alors, il suffit de placer les $k - 1$ autres boules dans les $n - 2$ premiers emplacements.

Vous penserez à traiter au début du code de votre fonction récursive tous les cas d'exclusion du raisonnement ci-dessus.

Votre code doit pouvoir calculer en quelques secondes $f(n, k)$ pour un entier $n \leq 35$. Voici quelques exemples d'appels de la fonction f :

- $f(4, 3)$ vaut 0 (il n'y a aucun rangement valide possible) ;
- $f(2, 0)$ vaut 1 (seul rangement confortable : $[[0, 0]]$)
- $f(2, 1)$ vaut 2, les rangements confortables sont $[[1, 0], [0, 1]]$
- $f(7, 3)$ vaut 10, ci-dessous tous les rangements confortables possibles :

1	[1, 0, 1, 0, 1, 0, 0]	[1, 0, 1, 0, 0, 1, 0]
2	[1, 0, 0, 1, 0, 1, 0]	[0, 1, 0, 1, 0, 1, 0]
3	[1, 0, 1, 0, 0, 0, 1]	[1, 0, 0, 1, 0, 0, 1]
4	[0, 1, 0, 1, 0, 0, 1]	[1, 0, 0, 0, 1, 0, 1]
5	[0, 1, 0, 0, 1, 0, 1]	[0, 0, 1, 0, 1, 0, 1]

Solution

Considérons un rangement confortable de k boules parmi n emplacements qu'on assimile à une liste L de n éléments valant 0 ou 1 où 0 signifie case vide et 1 case occupée. De deux choses l'une :

- OU BIEN la dernière case est vide et dans ce cas tous les rangements confortables sont obtenus en remplissant les $n - 1$ premières cases par des rangements confortables de k boules, ce qui en fait un nombre de $f(n - 1, k)$.
- OU BIEN la dernière case est occupée et dans ce cas tous les rangements confortables sont obtenus en remplissant les $n - 2$ premières cases par des rangements confortables de $k - 1$ boules (puisque une est déjà placée), ce qui en fait un nombre de $f(n - 2, k - 1)$.

Comme les deux cas précédents s'excluent et en même temps traitent toutes les situations possibles, on en déduit que la règle générale de calcul est $f(n, k) = f(n - 1, k) + f(n - 2, k - 1)$ et cela correspond au sous-problème.

Les cas de base correspondent à tous les cas où la relation précédente n'est pas valide (des arguments sont négatifs) et qui comprend les cas $n = 0$, $n = 1$ et $k = 0$. Si $n = 0$ il n'y a aucun rangement confortable. Si $n = 1$ tout rangement de $k \leq 1$ est confortable autrement dit, il y en a 1 dans chacun des deux cas possibles. Par ailleurs, pour que $f(n, k) = 0$ si $k > n$ (il y a plus de boules que d'emplacements).

```

1 def g(n, k):
2     if k>n:
3         return 0
4     if k==0:
5         return 1
6     if n==0:
7         return 0
8     if n==1:
9         return k
10    return g(n-1,k)+g(n-2,k-1)
11
12 print(g(35,12))

```

EXERCICES

Somme récursive

Ecrire une fonction récursive `somme(L, p)` qui calcule la somme des p premiers éléments d'une liste L d'entiers. Utiliser cette fonction pour calculer la somme des éléments d'une liste.

Produit des chiffres d'un entier

Observez d'abord cet exemple : prenons l'entier $n = 2016$ et écrivons la division entière de n par 10 sous la forme $n = 2016 = 10 \times 201 + 6$.

On remarque que cette écriture fait apparaître $u = 6$ qui est le chiffre des unités de n et $d = 201$ qui est le nombre obtenu à partir de n en supprimant le chiffre des unités. Plus généralement, tout nombre entier $n \geq 0$ s'écrit sous la forme $n = 10d + u$ où d est un entier positif et u est un entier tel que $0 \leq u \leq 9$. L'entier u est juste le *chiffre des unités* de n et d est juste le *nombre de dizaines* de n . Il est essentiel pour la suite de l'exercice de voir que d et u s'obtiennent facilement en posant la division entière de n par 10.

Ecrire une fonction récursive `produit` qui calcule le produit des chiffres d'un entier n . Par exemple, `produit(547256)` vaudra $5 \times 4 \times 7 \times 2 \times 5 \times 6 = 8400$.

Puissance récursive

Écrire une fonction récursive `power(a, m)` qui calcule a^m où a est un nombre et m un entier positif ou nul. Tester.

Test de croissance

On donne une liste non vide L d'entiers et on demande d'écrire une fonction récursive `estCroissante(L, i)` qui renvoie `True` si à partir de l'indice i , la liste L est triée dans l'ordre croissant et `False` sinon. Utiliser cette fonction pour tester si une liste non vide L d'entiers est triée ou pas dans l'ordre croissant.

Etre une puissance de 2

Les puissances de 2 soient les entiers de la forme 2^k où $k \geq 0$ est un entier ; les premières puissances de 2 sont :

1 2 4 8 16 32 64 128 256

Construire une fonction récursive `estPuissance2(n)` qui renvoie `True` si n (entier strictement positif) est une puissance de 2 et `False` sinon. Par exemple `estPuissance2(2048)` vaut `True` et `estPuissance2(2016)` vaut `False`

Indications pour construire la fonction récursive `estPuissance2(n)` :

- si n est impair et différent de 1 alors la fonction renvoie False
- si n est pair raisonner à l'aide de l'entier m tel que $n = 2m$.

Puissance de $2 \times$ nombre impair

Tout entier N peut s'écrire de manière unique comme un produit $N = 2^n d$ d'une puissance de deux et d'un nombre impair d . Par exemple, 2016 s'écrit : $2016 = 2^5 \times 63$ donc pour 2016, on a $n = 5$ et $d = 63$

On admettra et on utilisera que 2^n est la *plus grande* puissance de 2 qui soit un diviseur de N . On rappelle aussi que $1 = 2^0$ est une puissance de 2.

Écrire une fonction récursive `decomp(N)` qui renvoie la liste $[2^n, d]$.

Par exemple,

```
1 decomp(2016) -> [32, 63]
2 decomp(2017) -> [1, 2017]
3 decomp(64) -> [64, 1]
```

Indications pour construire la fonction récursive `decomp(N)` :

- le calcul de `decomp(N)` est facile si N est impair ;
- si N est pair, le calcul de `decomp(N)` se ramène à celui de la moitié de N .

Votre fonction doit pouvoir traiter un nombre entier ayant quelques dizaines de chiffres.

Suite de Syracuse en récursif

Soit la fonction f définie pour $n > 0$ entier par :

$$f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{sinon} \end{cases}$$

par exemple $f(13) = 40$ ou $f(10) = 5$.

- ① Coder en Python cette fonction. Tester avec $f(13)$ et $f(10)$.
- ② On itère la fonction f en partant d'un entier n . Par exemple, si $n = 13$, on obtient :

13 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

La conjecture de Syracuse affirme que si on itère f depuis n'importe quel entier $n > 0$ alors, on tombera forcément sur 1 (ce que l'exemple ci-dessus montre).

Écrire une fonction récursive `saut(x)` qui renvoie le nombre d'itérations pour arriver à 1 quand la suite commence avec x (par exemple, si $x = 13$ alors il y a donc 9 itérations pour arriver à 1).

Algorithme d'Euclide en récursif

On rappelle que le pgcd de deux entiers désigne le plus grand diviseur commun à ces deux entiers. Par exemple $\text{pgcd}(140, 100) = 20$: en effet, 140 et 100 sont des multiples de 20 et il n'existe aucun entier plus grand que 20 qui vérifie cette propriété.

L'algorithme d'Euclide permet de calculer de manière efficace le pgcd de deux entiers a et b . Il est basé sur l'observation suivante : si b est non nul, le pgcd de a et de b est aussi le pgcd de b et du reste de la division entière de a par b . Par exemple, comme $140 \% 100 = 40$, on $\text{pgcd}(140, 100) = \text{pgcd}(100, 40)$. On peut recommencer, ce qui donne $\text{pgcd}(100, 40) = \text{pgcd}(40, 20)$ et continuer $\text{pgcd}(40, 20) = \text{pgcd}(20, 0)$; cette fois on ne peut plus continuer puisque

$b = 0$ (et on ne peut pas diviser par 0). Mais $\text{pgcd}(a, 0)$ vaut toujours a et donc, ici, $\text{pgcd}(140, 100) = 20$. L'algorithme d'Euclide consiste donc à faire des divisions successives par le reste et à renvoyer le dernier reste non nul.

Ecrire une fonction récursive $\text{pgcd}(a, b)$ qui retourne le plus grand diviseur commun de deux entiers a et b par l'algorithme d'Euclide.

Palindrome en récursif

Ecrire une fonction récursive $\text{estSymetrique}(L, d, f)$ qui détermine si la liste des éléments d'une liste d'entiers L situés entre les indices d et f (indices inclus) est *palindromique*, autrement dit si elle est identique lorsqu'on la lit de la gauche vers la droite ou de la droite vers la gauche. Par exemple, $L = [81, 42, 13, 42, 81]$ est palindromique. Tester si une liste est palindromique.

Tours de Hanoï

On dispose de 3 tiges verticales A, B et C et de $n > 0$ disques percés en leur centre, de diamètres deux à deux distincts, empilés sur la tige A et disposés en sorte que chaque disque repose sur un disque de diamètre strictement supérieur. Le problème des tours de Hanoï est de translater cette pile sur la tige B par une succession de déplacements de disque d'une tige à une autre (on a besoin de la tige C), un seul à la fois, mais avec la contrainte qu'au cours des déplacements de disques, on ne pose jamais un disque sur un disque de diamètre strictement inférieur.

Ecrire une fonction récursive $\text{hanoi}(n, \text{source}, \text{but}, \text{temp})$ qui prend en entrée le nombre de disques et le nom des tiges et affiche chaque mouvement à effectuer (les noms des tiges de départ et d'arrivée) pour déplacer tous les disques depuis la position initiale (disques tous en A) à la position finale (disque tous en B).

Chiffres parmi 1 ou 8

Ecrire une fonction récursive $\text{chiffres81}(n)$ qui renvoie la liste de tous les entiers positifs ayant n chiffres et dont tous les chiffres sont parmi 1 ou 8. Par exemple, pour $n=3$, la fonction renvoie la liste suivante :

```
1 [111, 811, 181, 881, 118, 818, 188, 888]
```

On pourra utiliser que tout nombre à n chiffres s'écrit sous la forme $n = 10d + u$ où d est un nombre à $n - 1$ chiffres et u est le chiffre des unités ; par exemple, $818 = 81 \times 10 + 8$.

Chiffres -> nombre en base 10

On donne une liste L des chiffres d'un nombre entier $n \geq 0$ écrit en base 10, commençant par le chiffre des unités, un chiffre étant un entier entre 0 et 9. Ecrire une fonction *récursive* $\text{evalBase10}(L, i)$ qui à partir de cette liste L renvoie l'entier dont l'écriture en base 10 commence à l'indice i et se termine à la fin de la liste. Par exemple, $\text{evalBase}([5, 3, 6, 7, 2, 9], 2)$ renvoie le nombre 927635.

Ecrire une ligne de code utilisant $\text{evalBase10}(L, i)$ et qui renvoie la valeur de n , le nombre dont les chiffres sont dans L . Par exemple, si $L=[8, 3, 0, 2]$ alors $n=2038$.

Tri d'une liste par une méthode récursive

On veut trier⁶ dans l'ordre croissant une liste L d'entiers de la manière suivante : si L contient au moins $k \geq 2$ entiers, on trie la liste LL formée des $k - 1$ derniers éléments ; on appelle a le premier élément de L et b le premier élément de la liste triée LL ; si $a > b$ alors on échange les éléments a et b dans la liste L et on trie à nouveau la liste formée des $k - 1$ derniers éléments ; ainsi, la liste L sera bien triée.

Implémenter l'algorithme précédent en utilisant une fonction récursive $\text{tri}(L, d)$ où L est une liste d'entiers et où la fonction trie tous les entiers de L à partir de l'indice d . Pour obtenir

6. Vous allez voir, la méthode est vraiment ... idiote.

un tri de la liste L, on lancera donc `tri(L, 0)`.

Tester le tri pour une liste aléatoire de quelques centaines d'entiers. On utilisera le code ci-dessous :

```
1 def tri(L, d):
2     # Votre code
3
4
5 from random import randrange
6
7 N=600
8
9 L=[randrange(N) for _ in range(N)]
10 tri(L, 0)
```

Vous observerez que le code est relativement lent. Essayez d'améliorer la fonction ci-dessus. En effet, lorsqu'elle procède au 2^e tri, la liste à partir de son deuxième élément va être retriée inutilement. Pour parer à cela, changez légèrement la signature de la fonction en

```
1 def tri(L, d, ok)
```

où `ok` est un « drapeau » valant `True` ou `False` et qui indique si la liste L est oui ou non déjà triée à partir de son 2^e élément. Re-tester la fonction et observez une nette amélioration des performances (bien que le tri reste un tri dit *lent*).

Tirage du loto en récursif

Implémenter un tirage des 6 numéros du loto à l'aide d'une fonction récursive. Votre fonction aura la signature suivante

`loto(interdits, n)`

et elle renverra un tirage de `n` numéros entre 1 et 49 et tels qu'aucun numéro ne soit dans la liste des interdits. Le tirage du loto sera lancé par `loto([], 6)`.

Méthode square and multiply

- ① Ecrire une fonction récursive `power(a, n)` qui renvoie a^n en appliquant la méthode *square and multiply* qui consiste à ne faire que des succession d'élévation au carré ou de multiplication. Cette méthode consiste à remarquer que $a^{2k} = (a^k)^2$ et $a^{2k+1} = a(a^k)^2$.
Par exemple, pour calculer $x = 10^{36}$, on calculera $y = 10^{18}$ puis y^2 ; et pour calculer $y = 10^{18}$, on calculera $z = 10^9$ puis $y = z^2$; pour calculer $z = 10^9$, on remarquera que $9 = 2 \times 4 + 1$ et donc on calculera $u = 10^4$ et on obtiendra $z = 10 \times u^2$; et ainsi de suite.
- ② Modifier le code de la fonction pour qu'on puisse connaître le nombre de produits effectués. Combien faut-il de produit pour calculer 10^{2048} ?

Coefficient binomial en récursif

Ecrire une fonction récursive `binomial(n, p)` qui calcule le coefficient binomial « `p` parmi `n` » en utilisant la formule de construction du tableau de Pascal (chaque terme est somme des deux qui sont au-dessus de lui : $\binom{n}{p} + \binom{n}{p+1} = \binom{n+1}{p+1}$ valable pour tous entiers $n, p \geq 0$).

Juger du temps de calcul pour obtenir `binomial(32, 15)` (qui vaut 565722720).

Suite de Prouhet en récursif

La suite de Prouhet est la suivante :

```
1 0
2 01
3 0110
4 01101001
5 etc
```

Cette suite est une suite de chaînes composées de 0 et de 1 de la manière suivante : chaque chaîne T s'obtient à partir de la précédente U en adjoignant à U la suite V obtenue à partir de U en échangeant tout 0 en 1 et tout 1 en 0.

Ecrire une fonction récursive `prouhet(n)` qui renvoie la n -ème suite de Prouhet.

Liste des partages

On appelle partage de l'entier $s \geq 0$ toute liste croissante (au sens large) d'entiers strictement positifs et dont la somme vaut s . Par exemple, les partages de 6 sont les 11 listes suivantes :

```
1 1 1 1 1 1
2 1 1 1 1 2
3 1 1 1 3
4 1 1 2 2
5 1 1 4
6 1 2 3
7 1 5
8 2 2 2
9 2 4
10 3 3
11 6
```

Pour la suite de l'exercice, il est important de garder en tête qu'un partage est une **liste**. Par exemple, la liste suivante `[3, 6, 9, 11, 13]` est un partage de 42. Et donc si on cherche la liste de tous les partages de s , on obtiendra une liste de listes d'entiers.

Ecrire une fonction récursive `partage(k, s)` qui renvoie la liste de tous les partages de l'entiers s et dont le premier terme est supérieur ou égal à k . Par exemple, `partage(4, 17)` doit renvoyer une liste de 12 listes et qui sont représentées ci-dessous :

```
1 4 4 4 5
2 4 4 9
3 4 5 8
4 4 6 7
5 4 13
6 5 5 7
7 5 6 6
8 5 12
9 6 11
10 7 10
11 8 9
12 17
```

Afficher tous les partages de 13. Vérifier qu'il y a 53174 partages de 42.

Génération de toutes les parties d'un ensemble

Dans cet exercice, les ensembles sont codés sous forme de liste. Par exemple, l'ensemble A formé

des 4 éléments 81, 12, 31 et 65 sera codé sous la forme $A = [81, 12, 31, 65]$. L'ensemble ne contenant aucun élément sera noté $[]$.

Soit un ensemble A , de taille n . Programmer en Python la génération de toutes les parties de A autrement dit, écrire une fonction récursive `powerset(A)` qui renvoie la liste de toutes les parties de A en sorte que cette fonction renvoie une liste de liste. On rappelle qu'un ensemble ayant n éléments est formé de 2^n parties. Ainsi, `powerset([81, 12, 31])` est une liste formée des 8 listes ci-dessous :

```
1 []
2 [81]
3 [12]
4 [81, 12]
5 [31]
6 [81, 31]
7 [12, 31]
8 [81, 12, 31]
```

On notera que dans cette liste, figurent la partie vide (1^e ligne) et l'ensemble lui-même (dernière ligne).

La méthode pour construire `powerset(A)` consiste à observer que si A est une partie contenant un élément donné a et si on note B l'ensemble des éléments de A autre que a alors, toute partie de A est

- soit une partie de B ,
- soit une partie de B complétée de la partie (« le singleton ») contenant uniquement a .

Générer toutes les permutations récursivement

Une permutation d'un ensemble A correspond juste à une suite contenant chaque élément de A une fois et une seule. Par exemple, si A est l'ensemble $\{81, 12, 31, 65\}$ alors la suite $[81, 12, 31, 65]$ est une permutation de A mais $[12, 31, 65, 81]$ en est une autre, de même que $[12, 31, 81, 65]$.

Ecrire une fonction récursive `perm(A)` qui génère toutes les permutations d'un ensemble A formé de n éléments. L'ensemble A sera implémenté via une liste de n éléments. Toute permutation sera implémentée comme une liste d'éléments de A . Une permutation p étant donnée, raisonner en fonction de l'élément de la dernière position dans p .

Générer toutes les parties ayant un nombre d'éléments donné

Générer toutes les parties à p éléments d'un ensemble A donné de n éléments. Pour cela, on implémentera l'algorithme récursif naïf suivant : on se donne x dans A et on observe qu'une partie B de A ayant p élément est obtenue :

- soit à partir de x et d'une partie à $p - 1$ éléments ne contenant pas x
- soit d'une partie à p éléments ne contenant pas x .

Quelle est l'énorme limitation de cette méthode ?

Chapitre V

Calcul matriciel, exceptions

a) Notion de matrice

Une matrice représente un objet mathématique contenant des objets (en général des nombres) et tel que chaque objet de la matrice soit accessible à l'aide d'un couple d'indices entiers. En pratique, on se représente une matrice par un tableau 2D.

Ainsi, on peut représenter une matrice M à coefficients entiers ayant 2 lignes et 3 colonnes par le tableau :

$$M = \begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix}$$

En mathématiques, l'usage est de placer des parenthèses autour du tableau. En mathématiques encore, l'élément de la matrice M placé à la i -ème ligne et à la j -ème colonne est noté $M_{i,j}$ avec un double indice. Par exemple, avec la matrice ci-dessus, on a $M_{2,3} = 42$ qui est l'élément à la deuxième ligne et troisième colonne. Dans un contexte mathématique, l'indexation des lignes commence à 1 et non à 0.

Conventionnellement, étant donné une matrice ou encore un terme d'une matrice, on donne d'abord l'indice de ligne PUIS l'indice de colonne. Sauf contexte particulier, je désignerai toujours le nombre de lignes d'une matrice M par n et le nombre de colonnes par p où n, p sont des entiers strictement positifs et on dit alors que M est de taille $n \times p$. Ci-dessus, la matrice donnée en exemple est de taille 2×3 .

Ce qui distingue conceptuellement les matrices de simples tableaux 2D est qu'on associe à une matrice un certain nombre d'opérations mathématiques comme l'addition ou le produit.

Les matrices sont des outils très utilisés dans certains domaines de l'informatique qui nécessitent de la théorie des graphes ou de la recherche opérationnelle.

Matrices égales

Deux matrices M et N sont considérées comme égales lorsque :

- elles ont le même nombre de lignes n
- elles ont le même nombre de colonnes p
- pour chaque indice $i \in \{1, \dots, n\}$ et $j \in \{1, \dots, p\}$ on a $M_{i,j} = N_{i,j}$.

b) Comment implémenter des matrices en Python ?

Une matrice représente une structure de données. Une matrice se code aisément dans tout langage où on dispose d'une structure de données de type *array* comme les tableaux du C/C++ ou les listes en Python.

En pratique, les matrices seront à coefficients entiers, à coefficients flottants ou encore à coefficients booléens.

Matrice comme liste de listes

On rappelle qu'en Python, les éléments d'une liste peuvent être de type arbitraire, en particulier peuvent être encore des listes. Nous représenterons en Python une matrice ayant n lignes et p colonnes par une liste de n listes, chacune des précédentes listes ayant p éléments.

Par exemple, la matrice

$$M = \begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix}$$

sera représentée par le code Python suivant :

```
1 M = [[13, 12, 31], [81, 77, 42]]
2 print(M)
```

qui affiche

```
1 [[13, 12, 31], [81, 77, 42]]
```

Si la matrice est de taille $n \times p$, il est essentiel d'observer que tout élément de M est accessible par un couple (i, j) d'entiers tels que $0 \leq i < n$ et $0 \leq j < p$; noter les inégalités larges à gauche et strictes à droite et noter qu'en programmation, les indices commencent à 0. Observer que i est dans `range(n)` et j est dans `range(p)`; l'élément de M à l'indice (i, j) est `M[i][j]`, par exemple :

```
1 M = [[13, 12, 31], [81, 77, 42]]
2 print(M[1][2])
```

qui affiche

```
1 42
```

— Ligne 2 : l'élément à la 2^e ligne (d'indice 1) et à la 3^e colonne (d'indice 2).

On observera que la ligne d'indice `i` de `M` est juste la liste `M[i]` :

```
1 M = [[13, 12, 31], [81, 77, 42]]
2 print(M[1])
```

qui affiche

```
1 [81, 77, 42]
```

Les colonnes d'une matrice ne sont pas aussi *immédiatement* accessibles que les lignes le sont.

c) Nombre de lignes et nombre de colonnes

Soit une matrice M représentée comme ci-dessus, par une liste de listes. Alors,

- le nombre n de lignes de M est le nombre d'éléments de la liste `M`, à savoir `len(M)`
- le nombre de colonnes p de M est le nombre d'éléments de n'importe quelle ligne et comme une matrice contient au moins une ligne, p vaut `len(M[0])`.

Voici un exemple :

```

1 M = [[13, 12, 31], [81, 77, 42]]
2 n = len(M)
3 p = len(M[0])
4 print("Nombre de lignes ->", n)
5 print("Nombre de colonnes ->", p)

```

qui affiche

```

1 Nombre de lignes -> 2
2 Nombre de colonnes -> 3

```

Comme dans le code précédent, dans beaucoup de programmes travaillant avec une matrice M , il sera plus clair de définir et mémoriser dès le début du programme le nombre de lignes et le nombre de colonnes de M .

d) Parcours d'une matrice

Typiquement, lorsque l'on parcourt ligne par ligne une matrice M ayant n lignes et p colonnes, on utilise deux boucles **for** imbriquées, par exemple :

```

1 M = [[13, 12, 31], [81, 77, 42]]
2 n = len(M)
3 p = len(M[0])
4 for i in range(n):
5     for j in range(p):
6         print(M[i][j])
7     print("-" * 15)

```

```

8 13
9 12
10 31
11 -----
12 81
13 77
14 42
15 -----

```

Ce schéma est absolument fondamental pour coder du calcul matriciel.

En mathématiques, il est d'usage de nommer i l'indice de ligne courant et j l'indice de colonne courant. Lorsque c'est possible, je vous conseille d'utiliser cette convention quand vous codez du calcul matriciel.

e) Quelques matrices remarquables

Voici quelques matrices remarquables ainsi que leur codage en Python.

Matrice-ligne

Une *matrice-ligne* est simplement une matrice de taille $1 \times p$. Exemple de codage en Python :

```

1 M = [[13, 12, 31]]

```

ce qui mathématiquement correspond à la matrice

$$M = \begin{pmatrix} 13 & 12 & 31 \end{pmatrix}$$

Matrice-colonne

Une *matrice-colonne* est simplement une matrice de taille $n \times 1$. Exemple de codage en Python :

```
1 M = [[13, 12, 31]]
```

— Noter les doubles crochets.

ce qui mathématiquement correspond à la matrice $M = \begin{pmatrix} 13 \\ 12 \\ 31 \end{pmatrix}$.

Matrice nulle

C'est une matrice dont tous les coefficients sont nuls. Voici le codage en Python d'une fonction qui renvoie la matrice nulle de taille $n \times p$:

```
1 def matriceNulle(n, p):
2     "Constructeur de matrice de dimensions données"
3     M=[]
4     for i in range(n):
5         L=[]
6         for j in range(p):
7             L.append(0)
8         M.append(L)
9     return M
10
11 print(matriceNulle(3, 4))
```

qui affiche

```
1 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Cette fonction aura une grande importance dans les exercices puisqu'elle permet de créer un conteneur matriciel initialisé à 0, donc cette fonction (ce *constructeur*) sera appelée lorsqu'on a besoin de créer une nouvelle matrice dont la taille est connue.

Matrice carrée, matrice diagonale

Une *matrice carrée* est une matrice de taille $n \times n$ (autant de lignes que de colonnes) et on parle dans ce cas de matrice carrée d'ordre n .

On appelle *diagonale* d'une matrice carrée M la liste des coefficients situés sur la diagonale commençant en haut à gauche, à l'indice $(0,0)$, et se terminant en bas à droite, à l'indice $(n-1, n-1)$. Les termes de la diagonale de M sont les $M_{i,i}$. Cette diagonale est dite *principale*. L'autre diagonale est dite *secondaire*.

Le code ci-dessous affiche les termes diagonaux d'une matrice carrée :

```
1 M = [[13, 12, 31], [81, 77, 42], [10, 20, 30]]
2
3 for i in range(len(M)):
4     print(M[i][i])
```

```
5 13
6 77
7 30
```

Une matrice *diagonale* est une matrice carrée dont les coefficients hors de la diagonale principale sont nuls, par exemple $D = \begin{pmatrix} 42 & 0 \\ 0 & 81 \end{pmatrix}$

La matrice identité

La matrice identité d'ordre n est la matrice carrée d'ordre n , notée I_n , telle que :

- tous les termes diagonaux valent 1
- tous les termes non diagonaux valent 0

Par exemple, on a $I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

f) Somme, opposé, produit externe, transposée, trace

Somme de matrices

Étant donné deux matrices M et N de même taille (n, p) , on définit la matrice somme $S = M + N$ comme étant la matrice S telle que $S_{i,j} = M_{i,j} + N_{i,j}$. Par exemple,

$$\begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix} + \begin{pmatrix} -3 & 88 & 19 \\ -1 & 13 & 58 \end{pmatrix} = \begin{pmatrix} 10 & 100 & 50 \\ 80 & 90 & 100 \end{pmatrix}$$

Attention à ne pas utiliser sans précaution dans du code Python la notation `M + N` pour calculer la somme des matrices M et N .

Opposé

Étant donné une matrice M , on appelle *opposé* de M , noté $-M$, la matrice N telle que $N_{i,j} = -M_{i,j}$. Par exemple,

$$-\begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix} = \begin{pmatrix} -13 & -12 & -31 \\ -81 & -77 & -42 \end{pmatrix}$$

Attention à ne pas utiliser sans précaution dans du code Python la notation `-M` pour calculer l'opposé d'une matrice M .

Produit « externe »

Étant donné une matrice M et un nombre a , on appelle *produit externe* de a par M , noté aM , la matrice N telle que $N_{i,j} = a \times M_{i,j}$. Par exemple,

$$10 \begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix} = \begin{pmatrix} 130 & 120 & 310 \\ 810 & 770 & 420 \end{pmatrix}$$

Attention à ne pas utiliser sans précaution dans du code Python la notation `aM` pour calculer le produit externe du nombre a par une matrice M .

Transposée d'une matrice

Étant donné une matrice M de taille (n, p) , on définit la matrice N *transposée* de M comme étant la matrice de taille (p, n) dont les lignes sont exactement les colonnes de M autrement dit la matrice N telle que si $1 \leq i < n$ et $1 \leq j < p$ alors $N_{j,i} = M_{i,j}$. Par exemple, si

$$M = \begin{pmatrix} 13 & 12 & 31 \\ 81 & 77 & 42 \end{pmatrix} \text{ alors } N = \begin{pmatrix} 13 & 81 \\ 12 & 77 \\ 31 & 42 \end{pmatrix}.$$

Mathématiquement, on note souvent tM la transposée de M . Une matrice carrée est *symétrique* si elle est symétrique par rapport à sa diagonale principale, autrement dit si elle est égale à sa transposée. Par exemple, $M = \begin{pmatrix} 13 & 12 & 31 \\ 12 & 77 & 42 \\ 31 & 42 & 30 \end{pmatrix}$ est symétrique.

Trace d'une matrice carrée

On appelle *trace* d'une matrice **carrée** M le nombre noté $\text{tr}(M)$ qui est la somme des termes diagonaux de M . Par exemple, si $M = \begin{pmatrix} 13 & 12 \\ 77 & 42 \end{pmatrix}$ alors la trace de M vaut $13 + 42 = 60$.

g) Bibliothèque de fonctions matricielles

Dans les exercices de TP, vous allez devoir, entre autres, constituer une bibliothèque de fonctions matricielles telle que **somme**, **produitExterne**, **dimensions**, etc.

Plus précisément, vous allez devoir écrire des fonctions qui réalisent des opérations courantes sur les matrices comme la transposée, la somme ou le produit puis, au fur et à mesure, vous devrez placer le code des fonctions ainsi écrites dans un fichier (qui sera la *bibliothèque*) nommé **matrices.py** pour que ces fonctions soient réutilisables par du code Python. La réutilisation de code Python déjà écrit par le programmeur se fait un peu comme elle se fait en langage C via la directive `#include` sauf qu'en Python elle se fait avec une instruction d'importation qui ici aura la forme suivante : **from matrices import ***, cf. ci-dessous pour les détails.

En Python, toute unité de code susceptible d'être importé s'appelle un *module*. Un fichier Python est un cas particulier de module.

Le paragraphe suivant détaille le mécanisme d'importation du code Python d'un fichier depuis un autre fichier Python.

h) Fonctionnalités importées depuis un fichier

Depuis un fichier **b.py**, on peut appeler une fonction ou une variable définie dans un fichier **a.py**.

Soient les deux fichiers suivants **a.py** et **b.py** :

a.py

```
1 # a.py
2 def f(x):
3     return 10*x
4
5 def g(x):
6     return 100*x
```

b.py

```
7 # b.py
8 import a
9
10 print a.f(42)
```

— Ligne XX : le fichier **b.py** fait appel au code de **a.py**

On exécute le fichier **b.py** et on obtient l'affichage :

1 420

Autrement dit, le fichier Python `b.py` peut utiliser des fonctions écrites dans un autre fichier, ici `a.py`.

Pour utiliser dans le fichier `b.py` des fonctions définies dans le fichier `a.py` il faut que :

- `a.py` soit situé dans le même répertoire que celui de `b.py`
- `b.py` importe les fonctions à utiliser ce qui est fait avec l'instruction `import` ligne XX de `b.py` :

1 `import a`

Noter (cf. ligne XX) que le nom qui suit le mot-clé `import` est le nom du fichier sans son extension `.py`.

Pour appeler la fonction `f` depuis `b.py`, il faut alors préfixer le nom de `f` par le nom du module d'où on l'a importée (ligne XX) et en séparant avec un point :

1 `a.f(42)`

Un appel `f(42)` engendrerait une erreur :

`a.py`

```
1 # a.py
2 def f(x):
3     return 10*x
4
5 def g(x):
6     return 100*x
```

`b.py`

```
7 # b.py
8 import a
9
10 print(f(42))
```

```
5 File "b.py", line 4, in <module>
6     print(f(42))
7 NameError: name 'f' is not defined
```

i) La fonction d'affichage d'une matrice

Jusqu'à présent, pour afficher une matrice, on ne dispose que de la fonction `print` et l'affichage obtenu est très pauvre puisqu'il ne met pas en évidence les lignes ni les colonnes de la matrice et en outre, l'affichage contient des éléments parasites comme des virgules et des crochets :

```
1 M = [[13, 12, 31], [81, 77, 42]]
2 print(M)
```

qui affiche

```
1 [[13, 12, 31], [81, 77, 42]]
```

Si on veut afficher correctement une matrice, en laissant apparaître les alignements verticaux et horizontaux, il faut créer soi-même une fonction qui assure un formatage ad hoc de la matrice. Ci-dessous, voici une telle fonction :

```

1 def afficher(M):
2     "Affiche une matrice en respectant les alignements par colonnes"
3     w=[max([len(str(M[i][j])) for i in range(len(M))]) for j in range(len(M[0]))]
4     for i in range(len(M)):
5         for j in range(len(M[0])):
6             print("%*s" %(w[j],M[i][j]), end= ' ')
7         print()
8
9 A = [[42, 81, 1000], [2038, 10, 53], [-100, 84, -2020], [0, 3, 9]]
10 afficher(A)

```

qui affiche

1	42	81	1000
2	2038	10	53
3	-100	84	-2020
4	0	3	9

— Observer que dans l’affichage, les colonnes de la matrices sont bien alignées.

Dans les TP, on utilisera cette fonction sans chercher à savoir comment elle est codée.

On placera la fonction `afficher` dans le fichier `matrices.py` ce qui rendra la fonction prête à l’emploi par importation du module `matrices.py`.

j) Fonctions de la bibliothèque : exemples

On constitue un fichier `matrices.py` dont le contenu initial est :

matrices.py

```

1 def afficher(M):
2     "Affiche une matrice en respectant les alignements par colonnes"
3     w=[max([len(str(M[i][j])) for i in range(len(M))]) for j in range(len(M[0]))]
4     for i in range(len(M)):
5         for j in range(len(M[0])):
6             print("%*s" %(w[j],M[i][j]), end= ' ')
7         print()
8
9 def matriceNulle(n, p):
10     "Constructeur de matrice de dimensions données"
11     M=[]
12     for i in range(n):
13         L=[]
14         for j in range(p):
15             L.append(0)
16         M.append(L)
17     return M

```

Les fonctions décrites ci-dessous sont codées dans un autre fichier, par exemple, `exo.py` placé dans le même dossier que celui où se trouve `matrices.py`. Le fichier `matrices.py` est ensuite mis-à-jour (à la main, par le programmeur que vous êtes) de ces fonctions au fur et à mesure qu’elles sont écrites.

Fonction dimensions

Codons une fonction `dimensions` (noter le pluriel) qui prend en paramètre une matrice A et renvoie la liste des deux dimensions n et p de A :

```
1 from matrices import *
2
3 def dimensions(A):
4     return [len(A), len(A[0])]
5
6 A = [[42, 81, 1000], [2038, 10, 53], [-100, 84, -2020], [0, 3, 9]]
7
8 afficher(A)
9 print()
10 print(dimensions(A))
```

qui affiche

```
1   42 81 1000
2 2038 10   53
3 -100 84 -2020
4    0  3    9
5
6 [4, 3]
```

- Pour pouvoir utiliser la fonction `afficher` (ligne 8 du code Python), on doit importer le module `matrices`.
- La fonction renvoie directement la liste des dimensions, comme expliqué dans le paragraphe *Nombre de lignes et nombre de colonnes*
- Pour une meilleure lisibilité, on place un saut de ligne entre les deux affichages.
- On vérifie visuellement que la matrice A a 4 lignes et 3 colonnes.

Mettons-à-jour le fichier `matrices.py` :

matrices.py

```
1 def afficher(M):
2     "Affiche une matrice en respectant les alignements par colonnes"
3     w=[max([len(str(M[i][j])) for i in range(len(M))]) for j in range(len(M[0]))]
4     for i in range(len(M)):
5         for j in range(len(M[0])):
6             print("%*s" %(w[j],M[i][j]), end= ' ')
7         print()
8
9 def matriceNulle(n, p):
10    "Constructeur de matrice de dimensions données"
11    M=[]
12    for i in range(n):
13        L=[]
14        for j in range(p):
15            L.append(0)
16        M.append(L)
17    return M
18
19 def dimensions(A):
20    return [len(A), len(A[0])]
```

Fonction produitExterne

Codons une fonction produitExterne qui prend deux paramètres a et une matrice M et renvoie le produit aM :

```
1 from matrices import *
2
3 def produitExterne(a,A):
4     "Retourne le produit externe aA"
5     n, p = dimensions(A)
6     B=matriceNulle(n,p)
7     for i in range(n):
8         for j in range(p):
9             B[i][j]=a*A[i][j]
10    return B
11
12 A = [[5, 3, 1], [2, 1, 3], [-1, 8, -2], [7, 1, 4]]
13
14 afficher(A)
15 print()
16 afficher(produitExterne(10,A))
```

qui affiche

```

1  5  3  1
2  2  1  3
3 -1  8 -2
4  7  1  4
5
6 50 30 10
7 20 10 30
8 -10 80 -20
9 70 10 40

```

- Directement sous l'en-tête d'une fonction et à la même indentation que le corps de la fonction, on peut placer une chaîne de caractères ayant une valeur documentaire de la fonction. Ce type de chaîne documentaire s'appelle une *docstring*.
- Pour pouvoir utiliser aux lignes suivantes les fonctions `afficher`, `dimensions`, `matriceNulle`, on doit importer le module `matrices`.
- Le résultat à renvoyer (B) est créé par le constructeur `matriceNulle`
- Les coefficients de B sont écrasés par les valeurs souhaitées
- Pour une meilleure lisibilité de l'affichage, on place un saut de ligne entre les deux affichages de matrices

On peut alors mettre-à-jour le fichier de bibliothèques `matrices.py` en faisant un copier-coller du code de la fonction `produitExterne` dans le fichier `matrices.py`.

k) Tester son code avec des matrices

Il est capital de tester son code. Pour ce faire, on essaye de mettre son code en défaut. Il faut choisir un grand nombre d'exemples. Pour une meilleure lisibilité, on peut placer les données à tester dans une boucle `for`. Voici un exemple :

```

1 from matrices import *
2
3 def dimensions(A):
4     return [len(A), len(A[0])]
5
6 tests=[
7     [[1, 2],[3, 4]],
8     [[1, 2, -3],[3, 4, -1]],
9     [[1, 2, -3]],
10    [[1],[ -1]]
11    ]
12
13
14 for A in tests:
15     afficher(A)
16     print()
17     print(dimensions(A))
18     print("-"*10)

```

```

19 1 2
20 3 4
21
22 [2, 2]
23 -----
24 1 2 -3
25 3 4 -1
26
27 [2, 3]
28 -----
29 1 2 -3
30
31 [1, 3]
32 -----
33 1
34 -1
35
36 [2, 1]
37 -----

```

- On place les données à tester dans une liste `tests`. Noter que l'on choisit des matrices de toutes sortes, comme une matrice-ligne ou une matrice-colonne.
- On teste la fonction `dimensions` en parcourant la liste `tests` à l'aide d'une boucle `for`.
- On fait en sorte que les résultats soient le plus lisible possible : on affiche la matrice, on affiche le résultat, on place des séparateurs ou des sauts de ligne.

l) Produit de deux matrices

On va définir le produit de deux matrices M et N . Attention, cette opération est plus complexe que l'addition.

Pour faciliter le codage de la multiplication (cf. feuille de TP), les indices commenceront à 0 et non à 1.

Avant d'expliquer comment se calcule le produit $M \times N$ de deux matrices, il faut préciser que le nombre de colonnes de la première matrice M doit être égal au nombre de lignes de la deuxième matrice N . Ainsi, le produit d'une matrice de taille 4×2 et d'une matrice de taille 2×3 aura bien un sens mais pas le produit d'une matrice 2×3 et d'une matrice 2×3 .

Soient donc A une matrice de taille $n \times m$ et B une matrice de taille $m \times p$. On définit le produit $C := A \times B$ comme étant la matrice C de taille $n \times p$ telle que

$$c_{i,j} = a_{i,0} \times b_{0,j} + a_{i,1} \times b_{1,j} + \dots + a_{i,m-1} \times b_{m-1,j}$$

Ainsi

$$\begin{pmatrix} 10 & 40 & 20 & 30 \\ 20 & 70 & 50 & 90 \end{pmatrix} \begin{pmatrix} 3 & 1 & 5 \\ 2 & 4 & 7 \\ 8 & 0 & 6 \\ 1 & 5 & 4 \end{pmatrix} = \begin{pmatrix} 300 & 320 & 570 \\ 690 & 750 & 1250 \end{pmatrix}$$

Comment est obtenu le coefficient 570 du produit AB à la ligne d'indice 0 et la colonne d'indice

2 ? On examine la ligne $(10 \ 40 \ 20 \ 30)$ d'indice 0 de A et la colonne $\begin{pmatrix} 5 \\ 7 \\ 6 \\ 4 \end{pmatrix}$ d'indice 2 de B et on fait le calcul suivant :

$$(10 \ 40 \ 20 \ 30) \times \begin{pmatrix} 5 \\ 7 \\ 6 \\ 4 \end{pmatrix} = 10 \times 5 + 40 \times 7 + 20 \times 6 + 30 \times 4 = 570$$

Voilà pourquoi on dit que le produit matriciel s'effectue *ligne par colonne*.

Le produit matriciel dispose de bonnes propriétés telles que $(AB)C = A(BC)$ ou encore $A(B + C) = AB + AC$. Si A est une matrice de taille $n \times p$ alors $AI_p = A$ où I_p est la matrice identité d'ordre p . Attention cependant, deux exceptions notables : la propriété $AB = BA$ est rarement vraie et d'autre part, il se peut que AB soit la matrice nulle sans que pour autant A ou B soit nulle.

Grâce au produit matriciel, on peut définir la puissance A^m d'une matrice carrée A d'ordre n comme le produit de m matrices valant A . On convient que si $m = 0$ alors $A^m = I_n$ où I_n est la matrice identité.

m) Notion d'exception

En première approximation, une exception dans un programme Python est une situation qui survient lorsque

- le programme est en cours d'exécution
- l'interpréteur Python arrive à un moment du code où il est incapable de savoir comment l'exécuter.

Par exemple, le programme tente d'effectuer une division par zéro :

premiere_exception.py

```

1 print("Bonjour !")
2 print()
3
4 a = 42
5 b = 0
6 print(a/b)
7
8 print("Au revoir !")
9
10 Bonjour !
11
12 Traceback (most recent call last):
13   File "premiere_exception.py", line 6, in <module>
14     print(a/b)
ZeroDivisionError: division by zero
```

- Lignes 1-2 et 9-10 : le programme s'exécute normalement.
- Ligne 6 : le code effectue une division par zéro. C'est une opération mathématiquement impossible. L'interpréteur Python ne sait pas comment continuer l'exécution du code. On dit que Python lance une exception.

- Ligne 8 : Cette ligne n'est pas exécutée, le programme est suspendu
- Lignes 8-14 : le programme a planté ce qui peut se voir de différentes façons :
 - le mot **Au revoir !** (cf. ligne 8) n'apparaît pas dans l'affichage
 - le message d'erreur contient le mot **traceback** (bilan d'erreur)
- Ligne 14 : Le type de l'erreur est affiché ici **ZeroDivisionError** suivi d'une brève explication.

Le fait qu'un programme plante (le « crash ») est le pire qui puisse arriver car le programmeur ou l'utilisateur n'a plus aucun contrôle sur le programme et sur les données qu'il avait en mémoire : le programme est mort, on ne peut plus le ranimer, il n'est même pas dans le coma. Même si un programme doit ne pas exécuter ce pour quoi il est fait, on préférera toujours qu'il se ferme sans crash et que les données à sauvegarder soient sauvegardées voire que le programme soit ranimé.

Remarque importante sur le réalisme des exemples

On reprend le code ci-dessus :

premiere_exception.py

```

1 print("Bonjour !")
2 print()
3
4 a = 42
5 b = 0
6 print(a/b)
7
8 print("Au revoir !")

```

```

9 Bonjour !
10
11 Traceback (most recent call last):
12   File "premiere_exception.py", line 6, in <module>
13     print(a/b)
14 ZeroDivisionError: division by zero

```

- Ligne 5 : en pratique, un code ne sera jamais aussi naïf : le programmeur qui écrit ce code sait bien que le programme va planter ; dans une situation plus réaliste, la variable **b** pourrait plutôt représenter un nombre fourni par l'utilisateur pendant l'exécution du programme et dont la valeur est inconnue au moment où le code Python est écrit.

Voici donc un exemple de code plus réaliste :

```

1 print("Bonjour !")
2 print()
3
4 a = input("Entrez un nombre : ")
5 b = input("Entrez encore un nombre : ")
6 print()
7
8 print(("Le quotient vaut :", int(a)/int(b)))
9
10 print("Au revoir !")

```

```

11 Bonjour !
12
13 Entrez un nombre : 5
14 Entrez encore un nombre : 0
15
16 Traceback (most recent call last):
17   File "premiere_exception_plus_realiste.py", line 8, in <module>
18     print(("Le quotient vaut :", int(a)/int(b)))
19 ZeroDivisionError: division by zero

```

- Lignes 4-5 et 13-14 : l'utilisateur est invité à entrer des nombres.
- Ligne 8 : une opération interdite est effectuée.

n) La gestion des exceptions

Chaque fois que l'interpréteur Python rencontre une situation où il est incapable de poursuivre le traitement du code, l'interpréteur Python va lever une *exception*. Le langage Python donne la possibilité au programmeur de gérer les exceptions :

premiere_gestion_exception.py

```

1 print("Bonjour !")
2 print()
3
4 a = 42
5 b = 0
6
7 try:
8     print(a/b)
9 except Exception:
10    print("Une erreur est survenue")
11    print()
12
13 print("Au revoir !")

```

```

14 Bonjour !
15
16 Une erreur est survenue
17
18 Au revoir !

```

- Lignes 7-11 : la survenue d'une exception est gérée dans un bloc **try/except**
- Ligne 8 : l'interpréteur Python a levé une exception (une division par zéro).
- Lignes 14-18 : l'affichage est complet, pas de message **traceback** et l'instruction ligne 13 a été exécutée alors que postérieure à la levée de l'exception (ligne 8) : le programme n'a pas "planté".
- Ligne 9 : la capture de l'exception
- Lignes 10-11 : le traitement de l'exception pour éviter l'interruption définitive du programme.

Exécution d'un bloc try/except

La gestion des exceptions en Python se fait à l'aide de bloc `try/except` et appelé instruction `try`. Il s'agit d'une instruction composée contenant au moins deux blocs :

- une unique clause `try`
- une clause `except`.

Les noms `try` et `except` sont des mots-clés du langage Python.

Le corps de la clause `try` contient la partie de code susceptible de lever une exception.

La clause `except` s'appelle un *gestionnaire d'exception*. La clause `except` mentionne la catégorie d'exception qui doit être interceptée. Le type de l'exception, dans l'exemple ci-dessus, c'est l'exception de la catégorie `Exception` : ce nom est celui d'une exception « officielle » définie par le langage Python et ayant un caractère très général : pratiquement toute exception possible est du type `Exception`.

Comme une boucle `for` ou une instruction `return`, une instruction `try` est susceptible de modifier le flux d'exécution naturel d'un code Python. Si une exception est levée dans le corps de la clause `try` et si celle-ci est du même type que celle que la clause `except` cherche à capturer, l'exécution du bloc `try` est arrêtée là où l'exception a été déclenchée.

Une fois l'exception traitée le code se poursuit au-delà de la clause `except`.

Mise en garde

Attention, bien que Python gère les exceptions, Python ne fait aucun miracle ! Python ne vous donne la possibilité de gérer que les exceptions que vous avez prévues dans la clause `except`. Par exemple, dans le code ci-dessus, le programmeur a prévu, en utilisant le bloc `try/except` que le programme pouvait rencontrer une erreur de division par 0. Si le programmeur n'a rien prévu (pas de bloc `try/except`) , le programme lèvera une exception et le code se terminera par un plantage (interruption définitive du programme) et affichage d'un traceback dans la console.

D'autre part, gérer une exception ne règle pas forcément tous les problèmes qui peuvent survenir dans un programme. Ainsi, dans l'exemple ci-dessus, le calcul de division n'a pas été effectué (mais comment aurait-il pu l'être puisque, en l'état, l'opération est impossible?).

Nature d'une exception

Une exception est à la fois :

- une situation où l'exécution ne peut plus continuer
- un objet Python, objet au sens de la programmation orientée objet.

Lorsque l'exception, en tant que situation apparaît, un objet de type `Exception` est créé par l'interpréteur et auquel le code Python peut accéder ultérieurement afin par exemple obtenir des informations quant à la situation exceptionnelle voire pour y remédier.

o) La gestion d'une exception de type donné

```
1 print("Bonjour !")
2 print()
3
4 a = 42
5 b = 0
6
7 try:
8     print(a/b)
9 except ZeroDivisionError:
10    print("Une division par zéro est survenue")
11    print()
12
13 print("Au revoir !")
```

```
14 Bonjour !
15
16 Une division par zéro est survenue
17
18 Au revoir !
```

— Ligne 11 : lorsque le programmeur sait quel type d'erreur **peut** survenir, ici une erreur de division par zéro ligne 10, il fait suivre le mot-clé **except** du nom du type d'exception à capturer. Ici, le programmeur est supposé savoir qu'une exception de type division par zéro, en Python, s'appelle **ZeroDivisionError**. Une exception de type division par zéro est une exception par défaut (standard) du langage Python.

La liste complète des exceptions standard de Python figure dans la documentation de la bibliothèque standard de Python. Il en existe environ 50 parmi lesquelles une dizaine sont très courantes.

Nom d'une exception standard

Le nom d'une exception standard est à l'image du nom **ZeroDivisionError** : il se termine par **Error** avec une majuscule initiale et commence par un nom qui évoque le type d'erreur, la première lettre étant toujours une majuscule.

p) L'exception TypeError

```
1 print("Bonjour !")
2 print()
3
4 print("Bonjour" + 42)
5
6 print("Au revoir !")
```

```
7 Bonjour !
8
9 Traceback (most recent call last):
10   File "type_error.py", line 4, in <module>
11     print("Bonjour" + 42)
12 TypeError: Can't convert 'int' object to str implicitly
```

- Ligne 4 : L'addition est permise entre deux objets de type entier ou deux objets de type chaîne mais pas entre deux objets de types entier pour l'un et chaîne pour l'autre. Ici, l'interpréteur Python va lever une exception de type.
- Ligne 12 : le nom du type de l'exception est affichée, ici **TypeError**

Lorsqu'une instruction Python essaye de réaliser une opération pour des types non compatibles avec l'opération, il lève une exception de type `TypeError`.

q) Exception non gérée

`exception_non_capturee.py`

```

1 a = 42
2 b = 0
3
4 try:
5     print(a/b)
6 except TypeError:
7     print("Une erreur est survenue")
8     print()
9
10 print("Au revoir !")

```

```

11 Traceback (most recent call last):
12   File "exception_non_capturee.py", line 5, in <module>
13     print(a/b)
14 ZeroDivisionError: division by zero

```

- Lignes 4-5 : tentative d'exécution
- Ligne 5 : une exception de type **ZeroDivisionError** est levée puisque `b=0`.
- Ligne 6 : l'exception levée n'est pas de type `TypeError` donc la clause `except` ne capture pas l'expression. Il n'y a pas d'autre clause `except` donc l'exception se propage au de-là du bloc `try/except`. Comme aucun autre bloc `try/except` ne peut gérer l'exception, l'exception termine en traceback.

Si le nom de l'exception à capturer dans la clause `except`, disons **A**, ne correspond pas au type de l'exception, disons **B**, que le bloc `try` a levé, la clause `except` ne capturera pas l'exception **B** et l'exception se propagera pour, éventuellement terminer par un plantage du programme.

r) Plusieurs clauses `except`

Il se peut qu'une instruction ou un bloc d'instructions puisse générer plusieurs exceptions¹. Soit, dans un code Python, une expression telle que

```
1 print(a/b)
```

Elle pourra lever au moins deux types d'exception[[1]] :

- une exception `ZeroDivisionError` si `b` vaut 0
- une exception **TypeError** si par exemple, une des deux variables `a` ou `b` n'est pas un nombre, et par exemple, est de type chaîne, cf. les exemples ci-dessous

1. Toutefois, une seule exception peut être exécutée en même temps.

Le langage permet de capturer une exception parmi plusieurs possibles. Pour capturer les deux exceptions possibles, on peut envelopper l'instruction ci-dessus par l'instruction **try** suivante :

```
1 try:
2     print(a/b)
3 except ZeroDivisionError:
4     print("Tentative de division par zéro")
5 except TypeError:
6     print("Division de types incompatibles")
```

On observe que deux clauses **except** apparaissent, une pour chaque type d'exception susceptible d'être générée.

Voici plusieurs exemples basés sur le code précédent.

Exemple 1

```
1 a = 42
2 b = 0
3
4 try:
5     print(a/b)
6 except ZeroDivisionError:
7     print("Tentative de division par zéro")
8 except TypeError:
9     print("Division de types incompatibles")
10 Tentative de division par zéro
```

- Lignes 6-7 et 8-9 : la division par 0 est capturée
- Lignes 10-11 : cette clause est ignorée car l'erreur a déjà été capturée

Exemple 2

```
1 a = 42
2 b = "Bonjour"
3
4 try:
5     print(a/b)
6 except ZeroDivisionError:
7     print("Tentative de division par zéro")
8 except TypeError:
9     print("Division de types incompatibles")
10 Division de types incompatibles
```

- Ligne 7 : Une exception de type `TypeError` est capturée.
- Lignes 8-9 : cette clause est ignorée car l'erreur n'est pas de type `ZeroDivisionError`. L'examen se poursuit dans les clauses en dessous.
- Lignes 10-11 : l'exécution du code se poursuit dans ce bloc puisque l'exception est bien du type `TypeError`

Exemple 3

```
1 a = "Bonjour"
2 b = 0
3
4 try:
5     print(a/b)
6 except ZeroDivisionError:
7     print("Tentative de division par zéro")
8 except TypeError:
9     print("Division de types incompatibles")
```

```
10 Division de types incompatibles
```

- Ligne 7 : face à l'opération impossible "Bonjour"/0, il se trouve que Python lève une exception de type **TypeError** et non **ZeroDivisionError**.
- Lignes 8 : La 1^{er} clause est testée et ignorée car ne correspondant pas à l'exception levée
- Lignes 10-11 : L'exécution se poursuit dans cette clause.

s) Le rôle d'une clause except

Une clause except a pour but de remédier à une erreur pendant l'exécution pour éviter au moins le crash du programme. Face à la survenue d'une exception, le programme peut, en fonction des circonstances, réagir de façons variées, pouvant aller de la recherche d'un remède à la fermeture immédiate du programme. Voici un exemple :

```
1 a = int(input("Entrez un entier : "))
2 b = int(input("Entrez encore un entier : "))
3 print()
4 try:
5     q=a/b
6     print("Le quotient vaut : ", q)
7 except ZeroDivisionError:
8     print("Division impossible, le 2ème entier doit être non nul. Donnez un autre entier")
9     b= int(input())
10    print("Le quotient vaut :", a/b)
```

```
11 Entrez un entier : 42
12 Entrez encore un entier : 0
13
14 Division impossible, le 2ème entier doit être non nul. Donnez un autre entier :
15 10
16 Le quotient vaut : 4.2
```

- Ligne 7 : une exception est levée.
- Ligne 9 : l'exception est capturée
- Lignes 10-12 : La situation anormale est traitée : il est demandé à l'utilisateur de modifier sa valeur de **b**. Toutefois, ici, si l'utilisateur entre à nouveau une valeur nulle pour **b**, le programme crashera.

t) Accéder par un nom à une exception active

Suite au déclenchement d'une exception, un objet de type Exception est créé. Il est possible d'accéder à cet objet avec la mot clé **as** utilisé dans l'en-tête du gestionnaire d'exception :

```
1 a = 42
2 b = 0
3
4 try:
5     print(a/b)
6 except ZeroDivisionError as e:
7     print(type(e))
8     print(e)
9     print()
10 <class 'ZeroDivisionError'>
11 division by zero
```

- Division impossible
- une exception de type ZeroDivisionError est levée. Une instance de la classe ZeroDivisionError est créée à cette occasion. Dans la clause except, il est possible d'accéder à cette instance via la variable e.
- e est bien du type ZeroDivisionError.
- Quand on affiche une exception, on peut lire un message d'erreur adéquat.

u) except sec

Une instruction try/except peut contenir plusieurs clauses **except**

```
1 a = 42
2 b = 0
3
4 try:
5     print(a/b)
6 except ZeroDivisionError:
7     print("Tentative de division par zéro")
8 except TypeError:
9     print("Division de types incompatibles")
```

Le mot-clef **except** est suivi d'un nom de type d'exception. Mais, le **except** de dernière clause peut ne pas être suivi d'un nom de type d'exception :

```

1 a = 42
2
3 try:
4     print(a/b)
5 except ZeroDivisionError:
6     print("Tentative de division par zéro")
7 except TypeError:
8     print("Division de types incompatibles")
9 except:
10    print("Autre erreur")
11
12 print("FIN")

```

```

13 Autre erreur
14 FIN

```

- Lignes 3-6 : aucune variable `b` n'est déclarée.
- Ligne 6 : une exception de type `NameError` est levée.
- Lignes 7 et 9 : les clauses précédentes ne capturent pas une exception de type `NameError`.
- Ligne 11 : cette ligne capture toute exception qui n'a pas été capturée par les clauses précédentes.

Une telle clause est dite *attrape-tout* : sa fonction est d'empêcher l'interruption du programme même si une erreur se produit. Elle est utilisée :

- pour capturer une erreur d'origine inconnue
- pour capturer une erreur a priori d'origine prévisible (parce que le programmeur considère que c'est la seule exception qui pourrait apparaître) et que donc il est inutile de préciser.

Une telle clause peut rendre difficile la recherche d'une cause d'erreur d'un programme lorsque cette erreur est une erreur de codage de la part du programmeur.

2015

v) L'exception `TypeError`

```

1 print("Bonjour !")
2 print()
3
4 print("Bonjour" + 42)
5
6 print("Au revoir !")

```

```

7 Bonjour !
8
9 Traceback (most recent call last):
10   File "type_error.py", line 4, in <module>
11     print("Bonjour" + 42)
12 TypeError: Can't convert 'int' object to str implicitly

```

- Ligne 4 : L'addition est permise entre deux objets de type entier ou deux objets de type chaîne mais pas entre deux objets de types entier pour l'un et chaîne pour l'autre. Ici, l'interpréteur Python va lever une exception de type.

— Ligne 12 : le nom du type de l'exception est affichée, ici **TypeError**

Lorsqu'une instruction Python essaye de réaliser une opération pour des types non compatibles avec l'opération, il lève une exception de type `TypeError`.

w) Exception de type `ValueError`

Par défaut, Python lève une exception de type **ValueError** si un opérateur reçoit un argument du bon type mais dont la valeur est inappropriée. Par exemple, le type `int` renvoie un entier à partir d'un argument. Si cet argument est de type chaîne de caractères, il est attendu que la chaîne soit la représentation valide d'un entier en base 10. Donc si on donne à `int` un argument tel que `"bonjour"` ou `3.14` une exception sera levée :

```
1 z = "bonjour"
2 x = int(z)

3 Traceback (most recent call last):
4   File "_py", line 2, in <module>
5     x = int(z)
6 ValueError: invalid literal for int() with base 10: 'bonjour'
```

De même,

```
1 z = "2.0"
2 x = int(z)

3 Traceback (most recent call last):
4   File "_py", line 2, in <module>
5     x = int(z)
6 ValueError: invalid literal for int() with base 10: '2.0'
```

x) Exception de type `IndexError`

Une exception de type **IndexError** est levée chaque fois qu'un débordement d'indice dans une séquence se produit. Voici un exemple :

```
1 from random import randrange
2
3 JOURS_MOIS_NON_BISSEXTILE = [31, 28, 31, 30, 31, 30, 31, 31, 30,31,30,31]
4
5 for i in range(10):
6     m=randrange(0,20)
7     try:
8         nj=JOURS_MOIS_NON_BISSEXTILE[m-1]
9         print("Il y a", nj, "jours dans le", str(m)+"ème mois de l'année")
10    except IndexError:
11        print("Le", str(m)+"ème mois de l'année n'existe pas")
```

```

12 Il y a 31 jours dans le 1ème mois de l'année
13 Il y a 31 jours dans le 5ème mois de l'année
14 Il y a 31 jours dans le 1ème mois de l'année
15 Il y a 31 jours dans le 5ème mois de l'année
16 Le 15ème mois de l'année n'existe pas
17 Le 17ème mois de l'année n'existe pas
18 Il y a 30 jours dans le 9ème mois de l'année
19 Il y a 31 jours dans le 1ème mois de l'année
20 Il y a 31 jours dans le 1ème mois de l'année
21 Le 14ème mois de l'année n'existe pas

```

Dans le programme ci-dessus, on choisit au hasard un rang de mois entre 0 et 19 et on attend que le programme renvoie à l'aide de la liste prédéfinie `JOURS_MOIS_NON_BISSEXTILE` le nombre de jour que contient ce mois. Bien sûr comme il y a 12 mois dans une année, une exception va être levée par Python si le mois donné `m` a un rang valant au moins 13 puisque l'indice `m-1` est invalide.

y) Déclencher volontairement une exception

Face à des événements que l'interpréteur Python ne sait pas traiter, il déclenche une exception, de type appropriée. Mais le programmeur peut lui aussi volontairement déclencher une exception face à une situation qu'il aura identifiée. Par exemple, on veut écrire un programme qui calcule le nombre de secondes écoulées depuis minuit jusqu'à un moment de la journée exprimée avec trois variables `h`, `m` et `s`. Bien sûr cela suppose que par exemple `m` soit compris entre 0 et 59. Si ce n'est pas le cas, plutôt que de calculer des grandeurs peut-être absurdes, le programme va lever une exception :

```

1 def nb_sec(h,m,s):
2     if not (0<=h<24 and 0<=m<60 and 0<=s<60):
3         raise ValueError
4     return s+60*m+3600*h
5
6 h=8
7 m=72
8 s=13
9 for (h,m,s) in [[15, 19,42], [15, 73,42], [10, 0,0]]:
10     sec=nb_sec(h, m,s)
11     print("Il y a", sec, "secondes entre", h, "h", m, "m", s, "s et minuit")
12     print("-----")

```

```

13 Il y a 55182 secondes entre 15 h 19 m 42 s et minuit
14 -----
15 Traceback (most recent call last):
16   File "_py", line 10, in <module>
17     sec=nb_sec(h, m,s)
18   File "_py", line 3, in nb_sec
19     raise ValueError
20 ValueError

```

Noter que dans le programme ci-dessus, le premier moment de la journée est traité, que le deuxième lève une exception et que par conséquent le troisième n'ait pas traité par le programme.

Ici, le programmeur a choisi de déclencher une erreur de type **ValueError** car c'est la sémantique de ce genre d'erreur standard. Pour déclencher une erreur de type très général, on choisira pour type **Exception** (mais c'est déconseillé cela peut cacher une erreur du programmeur) :

```
1 def nb_sec(h,m,s):
2     if not (0<=h<24 and 0<=m<60 and 0<=s<60):
3         print("Données invalides")
4         raise Exception
5     return s+60*m+3600*h
6
7 h=8
8 m=72
9 s=13
10 for (h,m,s) in [[15, 19,42], [10, 0,0], [15, 73,42]]:
11     sec=nb_sec(h, m,s)
12     print("Il y a", sec, "secondes entre", h, "h", m, "m", s, "s et minuit")
13     print("-----")
```

```
14 Il y a 55182 secondes entre 15 h 19 m 42 s et minuit
15 -----
16 Il y a 36000 secondes entre 10 h 0 m 0 s et minuit
17 -----
18 Données invalides
19 Traceback (most recent call last):
20   File "_py", line 11, in <module>
21     sec=nb_sec(h, m,s)
22   File "_py", line 4, in nb_sec
23     raise Exception
24 Exception
```

Pour rendre la cause plus lisible, il est possible de donner en argument à l'exception levée un message plus explicite :

```
1 def nb_sec(h,m,s):
2     if not (0<=h<24 and 0<=m<60 and 0<=s<60):
3         raise ValueError("Nombre invalide d'heures, de minutes ou de secondes")
4     return s+60*m+3600*h
5
6 h=8
7 m=72
8 s=13
9 for (h,m,s) in [[15, 19,42], [15, 73,42], [10, 0,0]]:
10     sec=nb_sec(h, m,s)
11     print("Il y a", sec, "secondes entre", h, "h", m, "m", s, "s et minuit")
12     print("-----")
```

```

13 Il y a 55182 secondes entre 15 h 19 m 42 s et minuit
14 -----
15 Traceback (most recent call last):
16   File "_py", line 10, in <module>
17     sec=nb_sec(h, m,s)
18   File "_py", line 3, in nb_sec
19     raise ValueError("Nombre invalide d'heures, de minutes ou de secondes")
20 ValueError: Nombre invalide d'heures, de minutes ou de secondes

```

Bien sûr, comme toute exception, il est possible de la capturer avec un bloc **try/except** :

```

1 def nb_sec(h,m,s):
2     if not (0<=h<24 and 0<=m<60 and 0<=s<60):
3         raise ValueError
4     return s+60*m+3600*h
5
6 h=8
7 m=72
8 s=13
9 for (h,m,s) in [[15, 19,42], [15, 73,42], [10, 0,0]]:
10     try:
11         sec=nb_sec(h, m,s)
12         print("Il y a", sec, "secondes entre", h, "h", m, "m", s, "s et minuit")
13     except ValueError:
14         print("Données invalides")
15     print("-----")

```

```

16 Il y a 55182 secondes entre 15 h 19 m 42 s et minuit
17 -----
18 Données invalides
19 -----
20 Il y a 36000 secondes entre 10 h 0 m 0 s et minuit
21 -----

```

On constate ici que la 2^e liste était invalide mais que cela n'a pas interrompu le programme car l'exception levée a été capturée et traitée (par un message).

EXERCICES TYPE

Etre une matrice diagonale

Ecrire une fonction `estDiagonale(A)` qui teste si une matrice donnée `A` est une matrice diagonale (et en particulier carrée).

Solution

```

1 from matrices import *
2
3 def estDiagonale(A):
4     n,p=dimensions(A)
5     if n!=p:
6         return False
7     for i in range(n):
8         for j in range(n):
9             if i!=j and A[i][j]!=0:
10                 return False
11     return True
12
13
14 A=[[4, 0,0], [0, 5,0], [0, 0,7]]
15 B=[[4, 0], [0, 5], [0, 0]]
16
17
18 afficher(A)
19 print()
20
21 print(estDiagonale(A))
22
23 print("-----")
24 afficher(B)
25 print()
26
27 print(estDiagonale(B))

```

```

28 4 0 0
29 0 5 0
30 0 0 7
31
32 True
33 -----
34 4 0
35 0 5
36 0 0
37
38 False

```

Fonction somme de deux matrices

- ① Écrire une fonction `somme(A,B)` qui renvoie la somme de matrices A et B de même taille.
- ② Écrire une fonction `sontOpposees(A, B)` qui teste si deux matrices A et B sont opposées.

Solution

- ① Voici le code commenté de la fonction `somme(A,B)` :

```

1 from matrices import *
2
3 def somme(A,B):
4     n,p=dimensions(A)
5     S=matriceNulle(n,p)
6     for i in range(n):
7         for j in range(p):
8             S[i][j] = A[i][j] + B[i][j]
9     return S
10
11 A=[[-1, 4,2], [3, 5,-1]]
12 B=[[8, 6,-2], [1, -3,4]]
13
14 afficher(A)
15 print()
16
17 afficher(B)
18 print()
19
20 S= somme(A, B)
21 afficher(S)
22 -1 4 2
23 3 5 -1
24
25 8 6 -2
26 1 -3 4
27
28 7 10 0
29 4 2 3

```

- Ligne 1 : le fichier `matrices.py` doit être présent en compagnie du fichier courant pour pouvoir utiliser différentes fonctions déjà codées comme la fonction `afficher(M)` d’affichage d’une matrice M , la fonction `dimensions(M)` qui renvoie les dimensions de la matrice M ou la fonction `matriceNulle(n, p)`.
- Ligne 4 : Pour qu’on puisse faire la somme de deux matrices A et B , par définition, elles doivent avoir même dimensions, disons n et p .
- Ligne 4 : dans le code, on ne vérifie pas que les dimensions sont bien identiques.
- Ligne 5 : pour coder la somme, on commence par construire une matrice « vide » S ayant la même taille que A ou B et qui va recevoir les coefficients de $A+B$. Initialement S est remplie de zéros.
- Lignes 6-9 : ensuite, on parcourt les coefficients de S (lignes 6-7) et on remplace chaque coefficient par la somme des coefficients correspondants (ligne 8).

Le fichier `matrices.py` a pour contenu :

```

1  # matrices.py
2
3  def afficher(M):
4      "Affiche une matrice en respectant les alignements par colonnes"
5      w=[max([len(str(M[i][j])) for i in range(len(M))]) for j in range(len(M[0]))]
6      for i in range(len(M)):
7          for j in range(len(M[0])):
8              print("%*s" %(w[j],str(M[i][j])), end= ' ')
9              print()
10
11 def matriceNulle(n, p):
12     "Constructeur de matrice de dimensions données"
13     M=[]
14     for i in range(n):
15         L=[]
16         for j in range(p):
17             L.append(0)
18         M.append(L)
19     return M
20
21 def dimensions(A):
22     # renvoie le nombre de lignes et le nombres de colonnes de A
23     return [len(A), len(A[0])]

```

② Pour le test de matrices opposées, le code pourrait être :

```

1 from matrices import *
2
3 def somme(A,B):
4     n,p=dimensions(A)
5     S=matriceNulle(n,p)
6     for i in range(n):
7         for j in range(p):
8             S[i][j] = A[i][j] + B[i][j]
9     return S
10
11 def sontOpposees(A, B):
12     n,p=dimensions(A)
13     return sontEgales(somme(A,B), matriceNulle(n, p))
14
15 A=[[-1, 4,2], [3, 5,-1]]
16 B=[[1, -4,-2], [-3, -5,1]]
17 C=[[1, -4,-2], [-3, -5,42]]
18
19 afficher(A)
20 print()
21
22 afficher(B)
23 print()
24
25 afficher(C)
26 print()
27
28 print(sontOpposees(A,B))
29 print(sontOpposees(A,C))

```

```

30 -1 4 2
31 3 5 -1
32
33 1 -4 -2
34 -3 -5 1
35
36 1 -4 -2
37 -3 -5 42
38
39 True
40 False

```

Pour tester si deux matrices sont opposées, il suffit de regarder si leur somme est la matrice nulle. Donc on calcule leur somme avec la fonction `somme(A,B)` et on compare le résultat (avec la fonction `sontEgales` présent dans le fichier `matrices.py`) et la matrice nulle (créée avec la fonction `matriceNulle` du fichier `matrices.py`).

EXERCICES

Découverte : repérage dans une matrice

Définir explicitement une matrice A ayant 4 lignes et 3 colonnes et de coefficients de votre

choix (choisissez des coefficients négatifs comme positifs et de valeur variable (des grands et des petits)).

Afficher **A** avec la fonction **afficher** vue en cours.

Afficher le coefficient c de la matrice à la 4^e ligne et la 2^e colonne.

Mettre à la place du coefficient précédent c une valeur quelconque, par exemple 421.

Afficher la 2^e ligne de **A** (on rappelle qu'une ligne d'une matrice est juste un élément de la liste de listes que représente cette matrice).

Remplacer la 2^e ligne de **A** par la ligne $L = [12, 42, 81]$.

Découverte : matrice nulle

Définir, en utilisant la fonction **matriceNulle**, la matrice **N** ayant 4 lignes et 3 colonnes et dont tous les termes sont nuls puis afficher **N** avec la fonction **afficher** du cours.

Découverte : produit externe

Définir une matrice **A** ayant 4 lignes et 3 colonnes et de coefficients de votre choix. Appeler la fonction **produitExterne** pour calculer et afficher $10A$.

Fonction dimensions

Écrire une fonction **dimensions(A)** qui renvoie, sous forme de liste, le nombre de lignes et le nombre de colonnes de la matrice **A**.

Extraction d'une colonne

Écrire une fonction **extraireColonne(A, j)** qui renvoie, sous forme de liste, la colonne d'indice j de la matrice **A**.

Fonction trace d'une matrice

Écrire une fonction **trace(A)** qui renvoie la trace de la matrice carrée **A**.

Générer la matrice identité

Écrire une fonction **identite(n)** qui, à l'aide de la fonction **matriceNulle(n, p)** vue en cours, renvoie la matrice-identité de taille $n \times n$.

Matrices égales

Écrire une fonction **sontEgales(A, B)** qui teste si deux matrices **A** et **B** sont égales ie si

- **A** et **B** ont mêmes dimensions,
- les coefficients respectifs de **A** et de **B** sont égaux.

Générer une matrice aléatoire

Soient deux entiers $n, p > 0$ et soient deux entiers a et b avec $a \leq b$. Écrire une fonction **matrice_a_leat(n, p, a, b)** qui renvoie une matrice aléatoire de taille $n \times p$ et dont tous les coefficients sont des entiers de l'intervalle $[a, b]$ (utiliser la fonction **randrange** du module **random**).

Fonction transposée d'une matrice

- ① Écrire une fonction **transpose(M)** qui renvoie la transposée de la matrice **M**.
- ② Écrire une fonction **estSymetrique** qui teste si une matrice **M** est symétrique. La fonction **estSymetrique(M)** utilisera la fonction **transpose**. Une
- ③ Une matrice **M** est antisymétrique si elle vérifie $\mathbf{M} = -\mathbf{M}$ ou encore $\mathbf{M} + \mathbf{M} = \mathbf{0}$ (la matrice nulle). Écrire une fonction **estAntiSymetrique(M)** qui teste si une matrice **M** est antisymétrique. La fonction **estAntiSymetrique** utilisera la fonction **transpose** et la fonction **somme**.

Générer une matrice diagonale

On donne une liste L de n entiers. Ecrire une fonction `matriceDiagonale(L)` qui renvoie la matrice diagonale, cf. définition dans le Cours, dont la diagonale est constituée des éléments de la liste L (on utilisera la fonction `matriceNulle`).

Symétriser une matrice par rapport à sa diagonale secondaire

Soit une matrice carrée M . On appelle *diagonale secondaire* de M la suite de coefficients de M allant du coin inférieur gauche de M vers le coin supérieur droit de M . Par exemple, si M est la matrice

$$\begin{pmatrix} 5 & 3 & 2 \\ 6 & 4 & 0 \\ 7 & 1 & 9 \end{pmatrix}$$

la diagonale secondaire de M est 7, 4, 2.

On appelle *anti-transposée* d'une matrice carrée M la matrice N dont les coefficients sont les symétriques des coefficients de M par rapport à la diagonale secondaire. Par exemple, si M est la

matrice $\begin{pmatrix} 5 & 3 & 2 \\ 6 & 4 & 0 \\ 7 & 1 & 9 \end{pmatrix}$ alors son anti-transposée est $\begin{pmatrix} 9 & 0 & 2 \\ 1 & 4 & 3 \\ 7 & 6 & 5 \end{pmatrix}$

Ecrire le code d'une fonction `antiTranpose(M)` qui renvoie l'anti-transposée N de la matrice carrée M . Pour cela, on procédera comme suit :

- on construira une matrice nulle N de même taille que M ;
- soit une position (ligne, colonne) dans M d'indices (a, b) et soient (c, d) les indices de la position symétrique de (a, b) par rapport à la diagonale secondaire de M . Que valent alors $a + d$ et $b + c$ en fonction de la taille n de la matrice M ?

Implémenter le produit de deux matrices

Ecrire le code d'une fonction `produit(A,B)` qui renvoie la matrice produit AB . On supposera que la fonction `produit(A,B)` ne reçoit que des matrices A et B de tailles permettant d'effectuer leur produit. Tester avec les matrices suivantes :

$$A = \begin{pmatrix} -1 & 1 & -2 & 3 \\ -2 & -3 & 5 & -1 \end{pmatrix}, B = \begin{pmatrix} 3 & -1 & 5 \\ 2 & -4 & 1 \\ 3 & 0 & -3 \\ 4 & 5 & -4 \end{pmatrix}$$

et dont le produit vaut

$$\begin{pmatrix} 5 & 12 & -10 \\ -1 & 9 & -24 \end{pmatrix}$$

Tester intensément votre fonction. Pour cela vous testerez 1000 fois la validité l'identité

$$(A \times B) \times C = A \times (B \times C)$$

où les matrices A , B et C seront des matrices aléatoires à coefficients entiers entre -10 et 10 et de tailles également aléatoires entre 1 et 10. Pour cela, vous créerez une fonction de test unitaire `test_unitaire_produit` qui générera 3 matrices aléatoires A , B et C de tailles aléatoires entre 1 et 10 et testera si oui ou non, $(AB)C = A(BC)$ et une fonction de tests `test_produit(N)` où N est le nombre de tests à effectuer en appelant `test_unitaire_produit`.

Implémenter itérativement la puissance de matrice

Ecrire le code de la fonction `puissances_it(A, m)` qui renvoie A^m où A est une matrice carrée et m un entier positif, calculé par l'algorithme itératif. Tester la validité de votre code, en vérifiant que $A^7 A^8 = A^{15}$ où A est une matrice carrée aléatoire d'ordre 5. *On utilisera une fonction produit déjà codée.*

Générer un motif de matrice : équerres

On demande d'écrire une fonction $f(n)$ où $n > 0$ est un entier qui renvoie la matrice M suivante :

$$M = \begin{pmatrix} 0 & 1 & 2 & \dots & n-2 & n-1 \\ 1 & 1 & 2 & \dots & n-2 & n-1 \\ 2 & 2 & 2 & \dots & n-2 & n-1 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ n-2 & n-2 & n-2 & \dots & n-2 & n-1 \\ n-1 & n-1 & n-1 & \dots & n-1 & n-1 \end{pmatrix}$$

Explication : pour tout indice $k \in \{0, \dots, n-1\}$, les $k+1$ premiers éléments de la ligne d'indice k de M ainsi que les $k+1$ premiers éléments de la colonne d'indice k de M valent tous k .

Générer une matrice tridiagonale

On donne trois entiers a , b et c . On se donne aussi un entier $n > 0$. Ecrire une fonction `tridiag(a, b, c, n)` qui prend en paramètres a , b , c et n et qui renvoie la matrice carrée M d'ordre n (dite « tridiagonale ») dont

- la diagonale juste au-dessus de la diagonale principale est formée uniquement du coefficient a ,
- la diagonale principale est formée uniquement du coefficient b ,
- la diagonale juste en-dessous de la diagonale principale est formée uniquement du coefficient c ,
- les autres coefficients sont nuls

Par exemple, si $a = 4$, $b = 2$, $c = 1$ et $n = 4$ alors M est la matrice suivante

$$\begin{pmatrix} 2 & 4 & 0 & 0 \\ 1 & 2 & 4 & 0 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

On commencera par créer une matrice nulle M de taille $n \times n$ et on y placera ensuite les coefficients de la matrice à construire.

Remplir une matrice en diagonale

- ① Ecrire une fonction `remplirDiagonale(A, lig, col, k)`
 - qui remplit, par les entiers consécutifs à partir de l'entier k , la partie de la diagonale montante de A commençant à la position d'indices (lig, col)
 - qui renvoie la dernière valeur inscrite dans la diagonale.

Par exemple, si A est la matrice nulle de taille 8×9 alors `remplirDiagonale(A, 5, 2, 81)` renverra 86 et la matrice A sera remplie comme suit :

1	0	0	0	0	0	0	0	86	0
2	0	0	0	0	0	0	85	0	0
3	0	0	0	0	0	84	0	0	0
4	0	0	0	0	83	0	0	0	0
5	0	0	0	82	0	0	0	0	0
6	0	0	81	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

On utilisera une boucle **while** dont la condition garantira que la progression sur la diagonale reste bien à l'intérieur de la matrice.

- ② Écrire une fonction **remplirEnDiagonales(n, p)**, qui à partir de deux entiers $n, p > 0$, construit la matrice M de taille $n \times p$ et remplit, suivant ses diagonales montantes, par les entiers consécutifs $0, 1, 2, \dots, np - 1$.

Par exemple, si $n = 5$ et $p = 4$, la matrice M à obtenir doit s'afficher comme suit :

1	0	2	5	9
2	1	4	8	13
3	3	7	12	16
4	6	11	15	18
5	10	14	17	19

Trace d'une matrice et exception

Ecrire une fonction **trace(M)** qui renvoie la trace de la matrice M en supposant que celle-ci est carrée et qui lève une exception de type `ValueError` si la matrice contient des lignes de longueurs différentes.

Produit de matrices et exception

Ecrire une fonction **produit(A,B)** qui renvoie le produit de deux matrices mais qui gère la levée d'une exception en cas de fausses matrices (deux lignes d'une même matrice de longueurs différentes) ou des tailles incompatibles avec le produit matriciel.

Le jeu du plus/moins et gestion des exceptions

Les entrées en Python

Ci-dessous la fonction `capture_entier`

```

1 def capture_entier(msg):
2     return int(input(msg))
3
4 n = capture_entier("Entrez un nombre entier N : ")
5 print()
6
7 print("N x 10 =", 10 * n)
```

agit de la manière suivante :

- elle ouvre une console sous la cellule qui vous demande un entier
- vous entrez un entier dans la console
- vous validez avec ENTER
- la fonction capture votre entrée et la retourne

Testez pour comprendre.

Le jeu du devin

Le jeu du *plus/moins* oppose l'ordinateur (ie la machine) et le joueur (en principe, ce sera vous) et se déroule ainsi : la machine dispose d'un nombre secret x que le joueur doit découvrir. Après une invite de la machine, par exemple *Entrez votre choix*, le joueur propose un entier n tel que $1 \leq n \leq 100$. Le nombre est saisi au clavier, lisible dans une console sous la cellule de code et validé par le joueur en appuyant sur la touche ENTER. Ensuite, la machine réfléchit et répond :

- si $x > n$, la machine affiche : *C'est plus !*
- si $x < n$, la machine affiche : *C'est moins !*
- si $x = n$, la machine affiche *Bravo ! nombre trouvé en X essais* où X est le nombre de tentatives du joueur.

Quand l'entier n n'est pas le nombre-mystère, la machine propose à nouveau au joueur de fournir un nombre en affichant encore l'invite *Entrez votre choix*. On supposera que l'entier x est choisi, par le programme, aléatoirement entre 1 et 100.

Typiquement, voici ce qu'affichera une session de jeu, une fois terminée :

```
1 Vous devez decouvrir un nombre mystère compris entre 1 et 100
2 Entrez votre choix:
3 42
4 C'est PLUS
5
6 Entrez votre choix:
7 75
8 C'est MOINS
9
10 Entrez votre choix:
11 66
12 C'est MOINS
13
14 Entrez votre choix:
15 55
16 C'est PLUS
17
18 Entrez votre choix:
19 59
20 C'est MOINS
21
22 Entrez votre choix:
23 57
24 Bravo !
25 Nombre trouvé en 6 coup(s)
```

La capture du nombre entier se fera avec la fonction `capture_entier` ci-dessus. Coder un jeu du *plus/moins* (il n'est pas nécessaire d'écrire une fonction). Le déroulement du jeu se fait à l'intérieur d'une boucle `while` tant que l'utilisateur n'a pas trouvé le nombre mystère. Vous devez gérer les exceptions levées à cause d'une entrée du joueur incorrecte.