Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

MPSI·PCSI·PTSI
MP·PC·PSI·PT·TSI·TPC

j'intègre

EXERCICES INCONTOURNABLES

3e édition

DUNOD

Couverture : création Hokus Pokus, adaptation Studio Dunod

Pictogramme utilisé dans la table des matières : source Freepik

Retrouvez nos ouvrages pour les prépas scientifiques ici

http://dunod.link/prepassc

NOUS NOUS ENGAGEONS EN FAVEUR DE L'ENVIRONNEMENT :

Nos livres sont imprimés sur des papiers certifiés pour réduire notre impact sur l'environnement.

Le format de nos ouvrages est pensé afin d'optimiser l'utilisation du papier.

Depuis plus de 30 ans, nous imprimons 70 % de nos livres en France et 25 % en Europe et nous mettons tout en œuvre pour augmenter cet engagement auprès des imprimeurs français.

Nous limitons l'utilisation du plastique sur nos ouvrages (film sur les couvertures et les livres).

11 rue Paul Bert, 92240 Malakoff

www.dunod.com

ISBN 978-2-10-087629-7

Table des matières

Avant-propos

Cet ouvrage de la série « Exercices incontournables » traite de l'intégralité du nouveau programme d'informatique commune pour les deux années des différentes filières de classes préparatoires aux grandes écoles (sauf BCPST).

Les deux premiers chapitres reprennent la base de la programmation avec Python. Des rappels de cours et des exercices classiques vous permettront de vous familiariser avec la syntaxe Python.

Dans les exercices de certains chapitres (« Prise en main de Python », « Terminaison, correction, complexité », « Matrices de pixels et images, traitement d'images », « Dictionnnaire, pile, file, deque », « Graphes », « Intelligence artificielle et jeux à deux joueurs », « Bases de données »), vous trouverez un rappel de cours détaillé présentant le vocabulaire utilisé.

Pour chaque exercice classique, vous trouverez :

- La méthode de résolution expliquée et commentée étape par étape.
- Le corrigé rédigé détaillé.
- Les astuces à retenir et les pièges à éviter.

Les exercices signalés par le pictogramme ⊡ dans la table des matières sont à télécharger à partir de la page de présentation de l'ouvrage sur le site Dunod.

Vous y trouverez également tous les programmes Python des exercices, et des fichiers complémentaires sont fournis afin de tester les programmes, par exemple des images pour les exercices du chapitre 9 « Matrices de pixels et images, traitement d'images ».

https://dunod.com/EAN/9782100876297

1

Prise en main de Python

Exercice 1.1 : Assertion, moyenne, variance et écart-type d'une liste de nombres

On considère la liste de nombres : `L=[9, 10, 11, 20.5, 0, 12.0, -5, -8.3e1]`.

1. Écrire une fonction `rec_moy` qui admet comme argument `L` une liste non vide de nombres et retourne la moyenne de `L`.

2. Écrire une fonction `rec_variance` qui admet comme argument `L` une liste non vide de nombres et retourne la variance de la liste `L`.

3. Écrire une fonction `rec_ecart_type` qui admet comme argument `L` une liste non vide de nombres et retourne l'écart-type de la liste `L`.

4. Les en-têtes des fonctions peuvent être annotés pour préciser les types des paramètres et du résultat. Ainsi,

```
| def uneFonction(n:int, X:[float], c:str, u) -> list:
```

signifie que la fonction `uneFonction` prend quatre paramètres `n`, `X`, `c` et `u`, où `n` est un entier, `X` une liste de nombres à virgule flottante et `c` une chaîne de caractères ; le type de `u` n'est pas précisé. Cette fonction renvoie une liste.

Écrire une fonction `rec_moy2` qui admet comme argument `L` une liste non vide de nombres réels ou flottants et retourne la moyenne de la liste `L`. Annoter l'en-tête de la fonction pour préciser le type des données attendues en entrée et fournies en retour. Utiliser des assertions pour vérifier le type de `L`, le nombre d'éléments de `L` ainsi que le type des éléments de `L`.

Analyse du problème

On utilise une boucle `for` pour parcourir les différents éléments de la liste. On peut alors calculer la somme des éléments de la liste pour en déduire la valeur moyenne. Une assertion est une aide de détection de bugs dans les programmes. La levée d'une assertion entraîne l'arrêt du programme.

Cours :
L'installation de Python 3 peut se faire avec Pyzo.

Un script Python est formé d'une suite d'instructions. Une instruction simple est contenue dans une seule ligne. Si une instruction est trop longue pour tenir sur une ligne ou si on souhaite améliorer la lisibilité du code, le symbole « \ » en fin de ligne permet de poursuivre l'écriture de l'instruction sur la ligne suivante (voir corrigé 4).

Une affectation se fait avec l'instruction `n=3` : *n* prend la valeur 3 (voir exercice 1.4 « Affectation, objet immuable, copie »).

On peut utiliser « _ » dans le nom des variables mais pas « - ».

Le typage des variables est dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code. Dans l'exemple précédent, le type de `n` est `int`.

Le symbole dièse permet d'ajouter des commentaires dans les programmes Python : `# commentaire sur le programme`.

Le type d'une variable `n` s'obtient avec l'instruction : `type(n)`.

Les types de base des variables dans Python sont :

- Nombres entiers (positifs ou négatifs) : `int`
- Nombres à virgule flottante (ou nombres flottants) : `float`

  Exemple
  `a=-3.2e2` : $-3{,}2 \times 10^2 = -320$

- Booléens : `bool`
  Les variables booléennes sont `True` (vrai) et `False` (faux).

Opérations de base sur les entiers (`int`) : `+`, `-`, `*`, `//`, `**`, `%`

```
n=28
n//10      : 2 = quotient de la division euclidienne de n par 10
n%10       : 8 = reste de la division euclidienne de n par 10
n**3       : n puissance 3
```

Opérations de base sur les nombres flottants (`float`) : `+`, `-`, `*`, `/`, `**`

```
a=1/3      : a vaut 0.3333333333333333
2.6**(a)   : 2,6 puissance a
```

Comparaisons :

`a==b` : cet opérateur compare *a* et *b*. Si *a=b*, Python retourne `True`, sinon `False`
`a!=b` : *a* différent de *b*
`a>b`, `a<b`, `a>=b`, `a<=b` : strictement supérieur, strictement inférieur, supérieur ou égal, inférieur ou égal

Opérations sur les booléens (`bool`) :

```
or         : ou
and        : et
not        : non
```

```
rep1=True              # type bool. 2 valeurs possibles : True ou False
a=12
b=10
rep2=(a==12)and(b==20) # rep2=False pour le test : a=12 et b=20
rep3=not(a==13)        # rep3=True, on pourrait écrire : rep3 = a!=13
rep4=(a==12)or(b==20)  # True pour le test : a=12 ou b=20
```

Les types de base des conteneurs dans Python sont :

- Chaînes de caractères : `str`
  Structure indicée immuable. On ne peut pas modifier les éléments d'une chaîne de caractères.
  `s="C'est une phrase"` : utilisation de guillemets "
  `s1='phrase'` : utilisation d'apostrophes '
  `n=len(s)` : n=6 (nombre de caractères)
  `s1[0]` : le premier caractère de `s1` a pour indice 0 : `'p'`
  `s1[n-1]` : dernier caractère : `'e'`
  `C1='ab'`
  `C2=C1*3` : retourne `'abcabcabc'`, répétition de `'ab'`
  La chaîne de caractères `C1` est concaténée 3 fois avec elle-même.

- Listes : `list`
  Structure de données muable. On peut modifier les éléments d'une liste.
  `L=[]` : crée une liste vide
  `L.append(3)` : ajoute 3 **à la fin de la liste** `L`. On obtient `L=[3]`
  `L.append(2)` : ajoute 2 à la fin de la liste `L`. On obtient `L=[3, 2]`
  `x=L.pop()` : **supprime le dernier élément** (ici 2) de la liste `L`
  On récupère cet élément dans la variable `x`
  `L3=['r', 3, 'te']` : crée la liste `L3` contenant des chaînes de caractères et des entiers
  `len(L3)` : affiche la **longueur de la liste** `L3`
  Pour extraire des éléments d'une liste, voir exercice 1.6 « Slicing, extraction de tranche ».
  `L=[2*i for i in range(5)]` : on obtient `[0, 2, 4, 6, 8]` (**création par compréhension**)
  `L=[2,3]*3` : **répétition** de la liste `[2, 3]`
  On a alors `L=[2, 3, 2, 3, 2, 3]`
  `L=L+[5,8]` : **concaténation** : `L=[2, 3, 2, 3, 2, 3, 5, 8]`

Remarque :

Les indices des listes contenant *n* éléments sont numérotés de 0 à *n*–1 dans Python (idem pour les tuples et les deques). Pour obtenir le premier élément de la liste :

```
a=L[0]       # la variable a prend la valeur 2
```

- Tuples : `tuple`
  Les tuples sont des structures indicées immuables. Une fois le tuple créé, il ne peut pas être modifié. On peut créer un tuple avec ou sans parenthèses.
  `M=(2, 3, 8)` : crée le tuple `M`
  Ne pas confondre avec les listes, où on met des crochets.
  On crée le même tuple si on omet les parenthèses

`n=len(M)` : `n` = nombre d'éléments du tuple `M`
`M=2, 3, 9`
`x=M[0]` : récupère dans la variable `x` l'élément d'indice 0 du tuple `M`
`a, b, c=M` : dépaquette un tuple
On récupère dans chaque variable un élément du tuple.
Il faut connaître à l'avance le nombre d'éléments du tuple.
`M2=M+(2, 5)` : concaténation de deux tuples : `M2=(2, 3, 9, 2, 5)`
`M3=(2, 1)*3` : création avec répétition : `M3=(2, 1, 2, 1, 2, 1)`

- Deques : `deque`
  Une deque (se prononce « dèque ») est une structure de données muable qui généralise le fonctionnement des piles et des files (voir exercice 11.8 « Utilisation des deques »). On peut ajouter et supprimer des éléments aux deux extrémités. Pas d'extraction de tranche (ou slicing) avec les deques.
  `from collections import deque` : module permettant d'utiliser les deques
  `D=deque()` : création d'une deque vide `D`
  `D.append(3)` : ajoute 3 à l'**extrémité droite** de `D`
  `D.appendleft(5)` : ajoute 5 à l'**extrémité gauche** de `D`
  `x=D.pop()` : supprime l'élément à l'**extrémité droite** de `D`
  `x=D.popleft()` : supprime l'élément à l'**extrémité gauche** de `D`
  `D1=deque([3, 8, 5])` : création de la deque `D1`

```
for elt in D1:     # affichage de tous les éléments de la deque D1
    print(elt)
```

- Dictionnaires : `dict`
  Pour l'utilisation des dictionnaires, voir les exercices 11.1 « Opérations de base sur les dictionnaires » et 11.2 « Nombre d'occurrences de chaque élément d'une liste à l'aide d'un dictionnaire ».

Remarque : Pour le type `None`, voir la remarque de la question 2 dans l'exercice 10.1 « Tri par insertion ». On ne l'utilisera pas dans les autres exercices.

Il existe deux catégories d'objets dans Python :

- les objets dont la valeur peut changer sont dits muables (en anglais : mutable) : listes, dictionnaires, deques (voir chapitre 11 « Dictionnaire, pile, file, deque »)… ;
- les objets dont la valeur ne peut pas changer sont dits immuables (en anglais : immutable) : entiers, nombres flottants, booléens, chaînes de caractères, tuples…

Voir les exercices 1.4 « Affectation, objet immuable, copie » et 1.5 « Passage par référence pour les listes, effet de bord » pour les affectations et les arguments d'entrée des fonctions.

Quelques fonctions intrinsèques :

`abs(x)` : renvoie la valeur absolue de *x*
`int(x)` : convertit *x* en entier
`float(x)` : convertit *x* en flottant
`str(x)` : convertit *x* en chaîne de caractères
`bool(x)` : convertit *x* en booléen

Première utilisation de la boucle `for` :

```
x=5                  # affecte 5 à la variable x
for i in range(n):   # boucle faisant varier i de 0 inclus à n exclu
                     # ne pas oublier ':' à la fin de la ligne
    x=x+i            # incrémente x de i à chaque passage dans la boucle
                     # attention à l'indentation
```

Deuxième utilisation de la boucle `for` :

```
for elt in L:        # elt prendra successivement les éléments de L
    print(elt)       # L chaîne de caractères, liste, tuple, deque
                     # ou dictionnaire
```

Boucle `while` :

```
i=0
while i<=10:         # ne pas oublier : à la fin de la ligne while
    print(i)         # affichage de i
    i=i+1            # on incrémente i de 1 à chaque étape
```

La variable `i` est initialisée à 0 et incrémentée de 1 à chaque étape de la boucle `while`. On l'appelle un **compteur**.

Si la variable `i` est incrémentée d'une valeur différente de 1 ou décrémentée, on l'appellera un **accumulateur**.

Remarque : On peut utiliser l'instruction suivante :

```
    i+=1             # la variable i est incrémentée de 1
```

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante (voir exercice 10.6 « Tri à bulles »). Lorsqu'il y a plusieurs boucles imbriquées, l'instruction `break` ne fait sortir que de la boucle la plus interne.

Définition d'une fonction :

```
def f(x):      # définition de la fonction f ayant pour argument d'entrée x
               # ne pas oublier ':' à la fin de la ligne
    y=x+3      # y est une variable locale : elle est créée à l'appel
               # de la fonction et est détruite à la fin de la fonction
               # voir exercice 1.3 "Variables locales, variables globales"
    return y   # fin de la fonction et retourne la valeur y
               # attention à l'indentation
```

L'instruction `return` quitte la fonction même en cours d'exécution d'une boucle `for` ou `while`.

Structure conditionnelle :

```
if x==3:              # teste si x = 3
    y=3*x
elif x>3 and x<=4:    # si le test précédent n'est pas vérifié,
                      # alors teste si x >3 et si x <=4
    y=x+2
elif x>4 and x <5:    # si le test précédent n'est pas vérifié,
                      # alors teste si x >4 et si x <5
    y=x-2
else:                 # sinon (les tests précédents ne sont pas vérifiés)
    y=0
```

Importation de modules

Des fonctions traitant d'un même domaine sont regroupées dans des modules (par exemple les fonctions mathématiques `cos`, `sin`, `tan`… sont regroupées dans le module `math`). Différents modules peuvent être regroupés dans une bibliothèque. On utilise l'instruction `import module` pour importer un module.

```
import math          # importation du module math
```

Le module `math` contient des fonctions et des variables : `cos()`, `sin()`, `tan()`, `exp()`, `sqrt()` (racine carrée), `log()` (logarithme népérien), `log10()` (logarithme décimal), `pi` (nombre π)…

Pour utiliser les fonctions et les variables du module `math` :

```
a=math.pi/4
b=math.cos(math.pi/4)
c=math.sin(math.pi)
print(b)             # affiche 0.7071067811865476
print(c)             # affiche 1.2246467991473532e-16
```

Les nombres flottants ne permettent pas un calcul exact à cause de la représentation des nombres à virgule sur des mots de taille fixe. Un test du type `a==b` n'a en général pas de sens si `a` et `b` sont des nombres à virgule flottante. On remplacera donc ce test par : `abs(a-b)<epsilon` où `epsilon` est une valeur proche de zéro, choisie en fonction du problème à traiter et de l'ordre de grandeur des erreurs auxquelles on peut s'attendre sur `a` et `b`.

Ainsi, pour effectuer le test `sin(x)==0`, on n'utilisera pas l'instruction :

```
m.sin(x)==0          # si x=pi, ceci retourne pourtant False
```

mais les instructions suivantes :

```
eps=1**-8            # on choisit une valeur pour eps
abs(m.sin(x))<eps    # si x=pi, ceci retourne bien True
```

Lorsqu'on utilise l'instruction `from math import *`, il n'est plus nécessaire d'ajouter le nom du module pour utiliser ses fonctions :

```
from math import * # module math
a=pi/4
```

Certaines fonctions portent le même nom dans des bibliothèques différentes. Il est donc préférable de ne pas utiliser `from math import *` mais plutôt `import math`. On peut renommer le module `math` en `m` par exemple :

```
import math as m   # module math renommé m
a=m.pi/4
```

On peut importer des fonctions et des variables d'un module :

```
from math import cos, sin, tan, pi
a=cos(pi/4)
```

Voir exercice 1.4 « Affectation, objet immuable, copie » pour l'utilisation du module `copy`.

On utilisera la bibliothèque `PIL` dans le chapitre 9 « Matrices de pixels et images, traitement d'images ».

1. On considère une liste non vide `L`. On suppose que les éléments de la liste sont des nombres entiers ou flottants. On définit une variable `S` permettant de calculer la somme des éléments de la liste.

```
def rec_moy(L):
    # la fonction retourne la moyenne de la liste L
    S=0                     # initialisation de S à 0
    n=len(L)                # longueur de la liste L
    for i in range(n):      # i varie entre 0 inclus et n exclu
        S=S+L[i]            # on pourrait écrire S+=L[i]
    moyenne=S/n             # calcul de la moyenne
    return moyenne          # retourne la moyenne de L
```

Remarque :

La liste `L=[9, 10, 11, 20.5, 0, 12.0, -5, -8.3e1]` contient des entiers (`9`, `10`, `11`, `0`, `5`) ainsi que des nombres flottants (`20.5`, `12.0`, `-8.3e1`).

`12.0` est un nombre flottant (type `float`) alors que `12` est un nombre entier (type `int`).

`-8.3e1` est un nombre flottant alors que `83` est un nombre entier.

Cours :

Soit une liste de valeurs $X_1, X_2 \ldots X_N$. La moyenne des valeurs est définie par : $\langle X \rangle = \frac{1}{N}\sum_{i=1}^{N} X_i$.

La variance (ou écart quadratique moyen) est définie par : $\text{var}(X) = \langle X^2 \rangle - \langle X \rangle^2$ avec $\langle X^2 \rangle = \frac{1}{N}\sum_{i=1}^{N} X_i^2$. L'écart-type est défini par $\Delta X = \sqrt{\text{var}(X)}$.

2.

```
def rec_variance(L):
    # la fonction retourne la variance de la liste L
    S=0                     # initialisation de S à 0
    n=len(L)                # longueur de la liste L
    for i in range(n):      # i varie entre 0 inclus et n exclu
        S=S+L[i]**2         # on pourrait écrire S+=L[i]**2
    variance=S/n-rec_moy(L)**2
    return variance         # retourne la variance de L
```

3.

```
def rec_ecart_type(L):
    #la fonction retourne l'écart-type(float) de la liste L
    import math as m        # module math renommé m
    return(m.sqrt(rec_variance(L)))
```

Cours :

Une assertion est une aide de détection de bugs dans les programmes.

- La fonction `rec_moy` peut être appelée si le type de `L` est `list`. On ajoute la ligne suivante dans la fonction `def rec_moy2` :

```
| assert type(L)==list
```

Le programme teste si `type(L)==list`. Si la condition est vérifiée, le programme continue à s'exécuter normalement.
Si la condition `type(L)==list` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
assert type(L)==list AssertionError
```

- La fonction `rec_moy` peut être appelée si le nombre d'éléments de `L` est strictement positif. On ajoute la ligne suivante dans la fonction `def rec_moy2` :

```
| assert len(L)>0
```

Le programme teste si `len(L)>0`, sinon on aurait une division par 0 dans la fonction. Si la condition est vérifiée, le programme continue à s'exécuter normalement. Si la condition `len(L)>0` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
assert len(L)>0 AssertionError
```

On supprime les assertions dans la version finale du programme Python.

4.

```
def rec_moy2(L:list)->float:
    # la fonction retourne la moyenne (float) des éléments
    # de la liste L
    # On pourrait écrire : def rec_moy2(L:[float])->float:
    # L est une liste de nombres flottants
    assert type(L)==list
    assert len(L)>0
    S=0                 # initialisation de S à 0
    n=len(L)            # longueur de la liste L
    for i in range(n):  # i varie entre 0 inclus et n exclu
        assert type(L[i])==float or type(L[i])==int
        S=S+L[i]        # on pourrait écrire S+=L[i]
    moyenne=S/n         # calcul de la moyenne
    return moyenne      # retourne la moyenne de L
```

Si on exécute l'instruction `print('moyenne :',rec_moy2(3))`, Python affiche :

```
assert type(L)==list AssertionError
```

Si on exécute l'instruction `print('moyenne :',rec_moy2([]))`, Python affiche :

```
assert len(L)>0 AssertionError
```

Si on exécute l'instruction `print('moyenne :',rec_moy2([3, 4.0, 'a']))`, Python affiche :

```
assert type(L[i])==float or type(L[i])==int
AssertionError
```

Remarques :

On peut ajouter une chaîne de caractères dans l'instruction `assert` :

```
    assert type(L)==list, 'Le type de L doit être une liste.'
```

Le programme teste si `type(L)==list`. Si la condition est vérifiée, le programme continue à s'exécuter normalement.

Si la condition `type(L)==list` n'est pas vérifiée (on dit qu'on a une levée de l'assertion), alors le programme Python s'arrête et affiche le message d'erreur :

```
Le type de L doit être une liste.
```

Si une instruction est trop longue pour tenir sur une ligne ou si on souhaite améliorer la lisibilité du code, on peut utiliser le symbole « \ » en fin de ligne et poursuivre l'écriture de l'instruction sur la ligne suivante.

On peut écrire :

```
assert type(L[i])==float or type(L[i])==int, "Type de L[i] non correct."
```

ou

```
assert type(L[i])==float or type(L[i])==int,\
         "Type de L[i] non correct."
```

On peut ajouter des commentaires dans les programmes Python avec le symbole dièse. Pour ajouter un commentaire qui s'étend sur plusieurs lignes, on peut le commencer avec `'''` (trois apostrophes) ou `"""` (trois guillemets) et le terminer de la même façon :

```
'''
La fonction retourne la moyenne (float) des éléments de la liste L.
On pourrait écrire : def rec_moy2(L:[float])->float:
'''
```

Les assertions servent à tester des conditions critiques qui ne devraient jamais arriver. Ce sont des aides au développement des programmes.

Si ces erreurs (liste vide, type de `L` incorrect, type de `L[i]` incorrect) sont susceptibles d'arriver lors de l'exécution du programme final, alors il faut utiliser le test `if len(L)==0` et gérer par programmation l'erreur.

Exercice 1.2 : Boucle, test, fonction (banque PT 2015)

On pourra utiliser `L.reverse()` qui permet d'inverser les éléments de la liste `L`.

1. Soit l'entier $n = 1\,234$. Quel est le quotient, noté q, de la division euclidienne de n par 10 ? Quel est le reste ? Que se passe-t-il si on recommence la division euclidienne par 10 à partir de q ?

Écrire une fonction `calcul_base10` d'argument `n`, renvoyant une liste `L` contenant les restes des divisions euclidiennes successives.

La fonction vérifiera que `n` est un entier avec `assert`.

Écrire le programme principal demandant à l'utilisateur de saisir un entier `n` strictement positif et renvoyant la décomposition en base 10 de l'entier `n`.

2. Écrire une fonction `somcube`, d'argument `n`, renvoyant la somme des cubes des chiffres du nombre entier `n`. On pourra utiliser la fonction `calcul_base10`.

3. Écrire une fonction permettant de trouver tous les nombres entiers strictement inférieurs à 1 000 et égaux à la somme des cubes de leurs chiffres.

4. Écrire une fonction `somcube2` qui convertit l'entier *n* en une chaîne de caractères permettant ainsi la récupération de ses chiffres sous forme de caractères. Cette nouvelle fonction renvoie la chaîne de caractères ainsi que la somme des cubes des chiffres de l'entier *n*. On pourra utiliser la fonction `str` et manipuler les chaînes de caractères.

Analyse du problème

Cet exercice permet de s'entraîner à manipuler les fonctions, les boucles, les tests et les différents types rencontrés dans Python.

1. $n = 1\,234 = 123 \times 10 + 4$. Le reste vaut 4.

Si on recommence la division euclidienne de 123 par 10 : $123 = 12 \times 10 + 3$. Le reste vaut 3.

Si on recommence la division euclidienne de 12 par 10 : $12 = 1 \times 10 + 2$. Le reste vaut 2.

Si on recommence la division euclidienne de 1 par 10 : $1 = 0 \times 10 + 1$. Le reste vaut 1.

On obtient la décomposition en base 10 de n : 1, 2, 3, 4.

```
def calcul_base10(n):
    # la fonction renvoie une liste contenant les restes
    # des divisions euclidiennes successives
    assert type(n)==int
    L=[]   # création d'une liste vide
    while n>0:    # boucle tant que n > 0
        q=n//10   # quotient de la division euclidienne de n par 10
        r=n%10    # reste de la division euclidienne de n par 10
        L.append(r)   # on ajoute le reste dans la liste L
        n=q
    L.reverse()   # inverse l'ordre des éléments de la liste L
    return L      # retourne la liste L

# programme principal
n=int(input('Taper un entier strictement positif : '))
                  # conversion en entier du résultat de la saisie
```

```
print('Décomposition en base 10 : ',calcul_base10(n))
```

L'instruction `assert expression` de Python vérifie la véracité d'une expression booléenne et interrompt brutalement l'exécution du programme si ce n'est pas le cas.

2.

```
def somcube(n):
    # la fonction renvoie la somme des cubes des chiffres
    # de l'entier n
    somme=0              # initialisation de la variable somme
    L=calcul_base10(n)   # récupère la liste donnant
                         # la décomposition en base 10 de n
    for i in range(len(L)):   # i varie entre 0 inclus
                              # et len(L) exclu
        somme=somme+L[i]**3   # on ajoute L[i] à la puissance 3
                              # on peut écrire somme+=L[i]**3
    return somme

# programme principal
n=1234
print(somcube(n))             # affiche 100 pour n = 1234
```

3.

```
def affiche_liste_entier_cube():   # pas d'argument d'entrée
    # la fonction renvoie tous les nombres entiers strictement
    # inférieurs à 1000 et égaux à la somme des cubes
    # de leurs chiffres
    L=[]                      # création d'une liste vide
    for i in range(1000):     # i varie entre 0 inclus et 1000 exclu
        if i==somcube(i):     # teste si i est égal à la somme
                              # des cubes de ses chiffres
            L.append(i)       # ajoute i dans la liste L
    return L   # fin de la fonction et renvoie la liste L

# programme principal
print(affiche_liste_entier_cube())  # affiche
                                    # [0, 1, 153, 370, 371, 407]
```

4.

```
def somcube2(n):
    # la fonction convertit l'entier n en une chaîne de caractères
    # pour récupérer ses chiffres sous forme de caractères
    somme=0
    chaine=str(n)    # convertit n en une chaîne de caractères
    L=[]             # création d'une liste vide
    for elt in chaine:  # elt prend successivement
                        # les éléments de chaine
        L.append(elt)   # elt est un caractère que l'on ajoute
                        # dans L
        somme=somme+(int(elt))**3   # il faut convertir elt
                                    # en entier
    return L,somme      # on pourrait écrire return(L, somme)
```

```
# programme principal
n=int(input('Taper un entier strictement positif : '))
L1,res=somcube2(n)    # L1 contient la liste des chiffres de n
                      # res = somme des cubes des chiffres de n
```

Remarque :

La fonction `somcube2(n)` renvoie un tuple contenant deux éléments. Pour récupérer les éléments de ce tuple, on a plusieurs possibilités :

- Dépaquetage d'un tuple :

  La ligne `return L, somme` retourne un tuple : `(L, somme)`.

  Pour récupérer dans des variables séparées les éléments du tuple, on peut écrire :

```
L1, res=somcube2(25)
```

  On obtient alors : `L1=['2', '5']` et `res=133`.

- On définit un tuple *A* :

```
A=somcube2(25)
```

  Le tuple *A* vaut : `(['2', '5'], 133)`.

  Pour récupérer `['2', '5']`, le premier élément du tuple : `A[0]`.

  Pour récupérer `133`, le deuxième élément du tuple : `A[1]`.

Exercice 1.3 : Variables locales, variables globales

On considère le programme suivant :

```
def f() :
    global b
    print('d =', d)
    print('Premier print dans la fonction f : b =', b)
    a=3
    c=5
    b=b+c
    print('Deuxième print dans la fonction f : b =', b)
    print('Troisième print dans la fonction f : a =', a)
   return  # on pourrait supprimer cette ligne
           # ou écrire return None

a=2
b=2
d=3
print("Print avant l'appel de la fonction f : a =", a)
print("Print avant l'appel de la fonction f : b =", b)
f()
print("Print après l'appel de la fonction f : a =", a)
print("Print après l'appel de la fonction f : b =", b)
```

Qu'affiche Python lors de l'exécution du programme ? Analyser les différents affichages de `print`.

Analyse du problème

Ce programme permet de comprendre la différence entre les variables globales et les variables locales dans une fonction.

Cours :

Une variable locale est créée au début d'une fonction et est détruite lorsque la fonction est terminée. Elle existe uniquement dans le corps de la fonction.

Une variable globale est définie à l'extérieur d'une fonction. Le contenu de cette variable est visible à l'intérieur d'une fonction. L'instruction `global b` permet de définir la variable globale `b` dans la fonction `f`.

Le programme Python affiche :

```
Print avant l'appel de la fonction f : a = 2
Print avant l'appel de la fonction f : b = 2
d = 3
Premier print dans la fonction f : b = 2
Deuxième print dans la fonction f : b = 7
Troisième print dans la fonction f : a = 3
Print après l'appel de la fonction f : a = 2
Print après l'appel de la fonction f : b = 7
```

La variable `a` vaut toujours 2 après l'exécution de la fonction `f`.

Dans le corps de la fonction `f`, `a` est une variable locale qui n'a rien à voir avec la variable `a` définie dans le programme principal.

La variable `b` est modifiée par la fonction `f` car `b` est une variable globale (instruction `global b`). On retrouve 7 après l'appel de la fonction `f`.

La variable `c` est une variable locale. Elle n'est pas définie en dehors de la fonction. L'instruction `print(c)` en dehors de la fonction entraîne un message d'erreur de Python.

La variable `d` n'est pas définie dans la fonction `f`. Python cherche alors la valeur de `d` dans le programme principal. Python affiche alors : `d = 3`.

Remarque : L'instruction `global i,j` permet de désigner deux variables globales `i` et `j` dans une fonction.

Exercice 1.4 : Affectation, objet immuable, copie

La fonction `deepcopy(L)` du module `copy` permet de réaliser une copie profonde de la liste `L`.

1. Qu'affiche Python lors de l'exécution du programme suivant ?

```
i=3
j=i
print('Avant modification de i : i, j =', i, j)
i=5
print('Après modification de i : i, j =', i, j)
```

2. Qu'affiche Python lors de l'exécution du programme suivant ?

```
L1=[1, 3, 5, 7]
L2=L1
print('Avant modification de L2 : L1, L2 =', L1, L2)
L2[3]=2
L2[3]=print('Après modification de L2 : L1, L2 =', L1, L2)
```

3. Qu'affiche Python lors de l'exécution du programme suivant ?

```
L3=[1, 3, 5, 7]
import copy
L4=copy.copy(L3)
print('Avant modification de L4 : L3, L4 =', L3, L4)
L4[3]=2
print('Après modification de L4 : L3, L4 =', L3, L4)
```

4. Qu'affiche Python lors de l'exécution du programme suivant ?

```
L1=[1, 2, [3, 4], 5]
L2=L1
L3=copy.copy(L1)
L4=copy.deepcopy(L1)
L1[0]=12
L1[2][0]=30
print('L1 =', L1)
print('L2 =', L2)
print('L3 =', L3)
print('L4 =', L4)
```

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de copies de listes, deques et dictionnaires (voir chapitre 11 « Dictionnaire, pile, file, deque »).

Cours :

Il existe deux catégories d'objets dans Python :

- les objets dont la valeur peut changer sont dits muables (en anglais : mutable) : listes, dictionnaires, deques… ;
- les objets dont la valeur ne peut pas changer sont dits immuables (en anglais : immutable) : entiers, nombres flottants, booléens, chaînes de caractères, tuples…

Objets muables – Partage de valeurs par plusieurs variables

```
L1=[1, 2, 3, 4]
```

Cette affectation (ou assignation) est une instruction qui réalise les opérations suivantes :

- Création d'un objet muable (appelé `obj1`) de type `list` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj1` vaut : `[1, 2, 3, 4]`.
- Création de la variable `L1`.
- Association de la variable `L1` avec l'objet `obj1` contenant la valeur `[1, 2, 3, 4]`.

La variable `L1` ne contient pas `[1, 2, 3, 4]` mais uniquement la référence de l'objet `obj1`, c'est-à-dire l'adresse mémoire où est stocké `obj1`.

On peut modifier `[1, 2, 3, 4]` une fois que cet objet `obj1` de type `list` est créé. Les listes sont modifiables (muables).

Cours :

Contrairement à d'autres langages de programmation (C ou Java), une affectation dans Python est une association d'une variable avec un objet contenant la valeur. C'est le choix des concepteurs du langage Python.

```
L2=L1
```

L'instruction `L2=L1` n'affecte pas `[1, 2, 3, 4]` à `L2` mais réalise les opérations suivantes :

- Création du nom de variable `L2`.
- Affectation à la variable `L2` de la référence (ou adresse mémoire) où est stocké `[1, 2, 3, 4]`.

`L1` et `L2` font donc référence au même objet `[1, 2, 3, 4]`.

La copie est très rapide puisqu'on n'occupe pas deux fois plus de place mémoire.

Si on modifie `[1, 2, 3, 4]` via `L1`, alors cette modification sera également visible par `L2`.

```
L1[0]=10
```

On constate que `L2[0]` vaut 10 également. C'est tout à fait normal car `L1` et `L2` font référence à la même adresse mémoire de `[10, 2, 3, 4]`.

On ajoute un élément dans `L1` avec la fonction `append` :

```
L1.append(12)
```

L'élément 12 est ajouté dans `L1` et `L2` puisque `L1` et `L2` font référence à la même liste modifiable (ou muable) : `[10, 2, 3, 4, 12]`.

Comme dans les problèmes de concours, on utilise le langage suivant : le terme « liste » appliqué à un objet Python signifie qu'il s'agit d'une variable de type `list`. Idem pour les autres types : `int`, `float`, `bool`, `str`, `tuple`, `dict`, `deque`…

Objets immuables (non modifiables)

```
a=10
```

Cette affectation réalise les opérations suivantes :

- Création d'un objet immuable (appelé `obj2`) de type `int` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj2` vaut : `10`.

- Création de la variable `a`.
- Association de la variable `a` avec l'objet `obj2` contenant la valeur `10`.

La variable `a` ne contient pas `10` mais uniquement la référence de l'objet `obj2`, c'est-à-dire l'adresse mémoire où est stocké `obj2`.

On ne peut pas modifier `10` une fois que cet objet `obj2` de type `int` est créé. Les entiers sont non modifiables (immuables).

Cours :

```
b=a
```

Cette instruction n'affecte pas `10` à la variable `a` mais affecte la référence (ou l'adresse mémoire) où est stocké `10`.

```
a=11
```

Comme on ne peut pas modifier `10` (objet immuable), on crée un nouvel objet `11` avec une nouvelle adresse mémoire dans l'ordinateur. La variable `a` fait référence à l'adresse mémoire où est stocké `11`.

Par contre, `b` fait toujours référence à l'adresse mémoire où est stocké `10`.

```
print(b)    # b reste égal à 10
```

On retrouve le même résultat pour tous les objets immuables : entiers, nombres flottants, booléens, chaînes de caractères, tuples…

Deux cas peuvent se présenter après l'exécution de l'instruction `L2=L1` :

- `L1` est un objet muable (mutable, en anglais, par exemple liste, dictionnaire, deque…) : si on modifie `L1` dans la suite du programme, alors `L2` est également modifié puisque `L1` et `L2` font référence à la même adresse mémoire.
- `L1` est un objet immuable (immutable, en anglais, par exemple entier, nombre flottant, booléen, chaîne de caractères, tuple…) : si on modifie `L1` dans la suite du programme, alors `L2` n'est pas modifié.

Cours :

On rencontre deux catégories de copies pour les objets muables (listes, dictionnaires, deques…) :

- La fonction `copy()` réalise une copie superficielle. Les éléments sont copiés s'il n'y pas de structure imbriquée. Si les éléments sont des listes par exemple, alors l'adresse mémoire des listes est copiée.
- La fonction `deepcopy()` réalise une copie profonde pour les structures imbriquées. Si les éléments sont des listes, alors la copie profonde copie bien les listes imbriquées.

```
import copy
L2=copy.copy(L1)
```

Python exécute une copie superficielle de `L1`, c'est-à-dire qu'il crée une nouvelle liste `L2` en copiant tous les éléments de `L1` dans `L2` puisqu'ils ne contiennent pas de structure imbriquée. Dans ce cas, `L1` et `L2` ne font plus référence à la même adresse mémoire.

La copie superficielle s'applique également aux dictionnaires et deques (voir chapitre 11 « Dictionnaire, pile, file, deque »).

Remarque :

Avec certains langages (langage C++, Java par exemple), le typage des variables est statique, c'est-à-dire qu'il faut d'abord déclarer (ou définir) le nom et le type des variables et ensuite leur affecter (ou assigner) une valeur compatible avec le type déclaré.

Avec le langage Python, le typage des variables est dynamique : l'interpréteur détermine automatiquement le type qui correspond au mieux à la valeur fournie lors de l'affectation.

1. Python affiche :

```
Avant modification de i : i, j = 3 3
Après modification de i : i, j = 5 3
```

Les résultats affichés dans Python sont tout à fait prévisibles. On va voir dans la question 2 que la même syntaxe appliquée aux listes donne des résultats surprenants !

2. Python affiche :

```
Avant modification de L2 : L1,L2 = [1, 3, 5, 7]
                                   [1, 3, 5, 7]
Après modification de L2 : L1,L2 = [1, 3, 5, 2]
                                   [1, 3, 5, 2]
```

Le programme de la question 2 est exactement le même que celui de la question 1 sauf qu'on manipule des listes au lieu de manipuler des entiers. Le comportement est complétement différent : la liste `L1` a été modifiée !

L'instruction `L2=L1` n'a pas effectué une copie de `L1` dans `L2` mais a copié uniquement la référence de la liste, c'est-à-dire l'adresse mémoire de la liste. `L1` et `L2` font donc référence à la même adresse mémoire de l'ordinateur. Si on modifie un élément de `L2` alors `L1` est également modifié.

3. Python affiche :

```
Avant modification de L4 : L3,L4 = [1, 3, 5, 7]
                                   [1, 3, 5, 7]
Après modification de L4 : L3,L4 = [1, 3, 5, 7]
                                   [1, 3, 5, 2]
```

L'instruction `L4=copy.copy(L3)` permet de réaliser une copie superficielle de `L3`. Les listes `L3` et `L4` ont des adresses mémoire différentes.

La modification d'un élément de `L4` n'a donc aucune conséquence sur `L3` puisque les éléments ne contiennent de structure imbriquée.

4. Python affiche :

```
L1 = [12, 2, [30, 4], 5]
L2 = [12, 2, [30, 4], 5]
L3 = [1, 2, [30, 4], 5]
L4 = [1, 2, [3, 4], 5]
```

L'affectation `L2=L1` ne réalise pas une copie de `L1`. Si on modifie un élément de `L1`, alors cet élément est modifié dans `L2` puisque `L1` et `L2` pointent vers la même adresse mémoire.

L'instruction `L3=copy.copy(L1)` permet de réaliser une copie superficielle de `L1`. Si on modifie un élément de `L1` qui est une liste (exemple `[30, 4]`) alors cet élément est également modifié dans `L3`.

L'instruction `L4=copy.deepcopy(L1)` permet de réaliser une copie profonde de `L3`. Si on modifie un élément de `L1` qui est une liste (exemple `[30, 4]`) alors cet élément n'est pas modifié dans `L3`.

Exercice 1.5 : Passage par référence pour les listes, effet de bord

On considère le programme suivant :

```
def f(a, b, L):
    a+=1
    print('Print dans la fonction : a =', a)
    b=b+1
    print('Print dans la fonction : b =', b)
    print('Print dans la fonction : d =', d)
    L.append(4)       # on ajoute un élément dans L
    print('Print dans la fonction : L =', L)
    return a

a=3
b=5
d=3
L=[1, 2, 3]
c=f(a, b, L)
print('Print après la fonction : a =', a)
print('Print après la fonction : b =', b)
print('Print après la fonction : c =', c)
print('Print après la fonction f : L =', L)
```

Qu'affiche Python lors de l'exécution du programme ? Analyser les différents affichages de `print`.

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de l'utilisation de listes dans les arguments d'entrée des fonctions.

Cours :
Les arguments d'entrée des fonctions sont tous passés par référence. On considère deux cas :

- Objet muable (liste, dictionnaire, deque…) : toute modification de cet objet dans la fonction est visible en dehors de la fonction. C'est « l'effet de bord » puisque la fonction modifie des données définies hors de sa portée locale.
- Objet immuable (entier, nombre flottant, booléen, chaîne de caractères, tuple…) : toute modification de cet objet dans la fonction n'est pas visible en dehors de la fonction. Pour simplifier, tout se passe comme si ces objets étaient passés par valeur.

Si on modifie un argument d'entrée muable (liste, dictionnaire, deque…) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction.

Python affiche :

```
Print dans la fonction : a = 4
Print dans la fonction : b = 6
Print dans la fonction : d = 3
Print dans la fonction : L = [1, 2, 3, 4]
Print après la fonction : a = 3
Print après la fonction : b = 5
Print après la fonction : c = 4
Print après la fonction f : L = [1, 2, 3, 4]
```

La fonction `f` retourne la valeur `4` qui est affectée dans la variable `c`.

Avant l'appel de la fonction `f`, la variable `b` fait référence à `5`. La variable `b` est passée par référence dans la fonction `f` et fait toujours référence à `6`. L'instruction `b=b+1` dans la fonction `f` ne peut affecter `5` qui est immuable. La variable `b` dans la fonction `f` fait donc référence à une nouvelle valeur `6`. La variable `b` du programme principal garde donc la même valeur `6`.

Pour simplifier, cela revient à dire que les variables `a` et `b` sont passées par valeur : l'exécution de la fonction `f` évalue d'abord `a` et `b` puis exécute `f` avec les valeurs calculées `a` et `b`.

La variable `L` fait référence à la liste `[1, 2, 3]` qui est un objet muable. La liste `L` est passée par référence. Lorsqu'on modifie `L` dans la fonction `f`, on modifie la liste `[1, 2, 3]`. On dit que la liste `L` est modifiée en place dans la fonction.

La variable `L` dans la fonction `f` fait référence à la même adresse mémoire que la variable `L` dans le programme principal.

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction (variable `d` par exemple), Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du programme.

Remarque : Voir exercice précédent, « Affectation, objet immuable, copie », pour avoir une copie profonde avec la fonction `deepcopy()`.

Exercice 1.6 : Slicing, extraction de tranche

On considère le programme suivant :

```
L1=[i for i in range(2, 7, 2)]   # création par compréhension
print('L1 =', L1)
print(L1[:-1])
```

Qu'affiche Python lors de l'exécution du programme ?

Analyse du problème
La technique du slicing, ou extraction de tranche, permet d'extraire des éléments d'une liste.

Cours :
Lorsqu'on veut extraire des éléments d'une liste, d'un tuple ou d'une chaîne de caractères, on utilise la technique du slicing, ou extraction de tranche. Il suffit de mettre entre crochets les indices correspondant au début et à la fin de la « tranche ». On utilise la syntaxe :

```
L[start:stop]
```

`start` : indice de départ, inclus
`stop – start` : longueur de la liste extraite (avec un pas de 1 par défaut)
`stop` : indice final, exclu

Exemple pour `L=[3, 5, 1, 8, 10]` : `L[1:4]` renvoie `[5, 1, 8]`.

L'indice de départ vaut 1 avec `L[1]` = 5.
L'indice final vaut 4 – 1 = 3 avec `L[3]` = 8.
On récupère bien 3 éléments : `[5, 1, 8]`.

Il faut bien retenir l'instruction : `L[start:stop]`.
Les indices sont compris entre `start` inclus et `stop` exclu.
On peut retenir facilement que le nombre d'éléments vaut `stop – start` avec un pas de 1 par défaut.

Cours :
On peut utiliser l'extraction de tranche avec un pas différent de 1. On utilise alors la syntaxe suivante :

```
L[start:stop:step]
```

`start` : indice de départ, inclus
`stop` : indice final, exclu
`step` : cette variable désigne le pas.

Exemple pour `L=[3, 5, 1, 8, 10]` : `L[1:4:2]` renvoie `[5, 8]`.

L'indice de départ vaut 1 avec `L[1]` = 5.
L'indice final 4 est exclu.
On ne prend qu'un élément sur 2.

On peut omettre n'importe lequel des arguments dans `L[start:stop:step]`.

Par défaut : `start` = 0, `stop` = indice du dernier élément de la liste + 1, `step` = 1.

`L[:3]` renvoie tous les éléments dont les indices sont compris entre 0 inclus et 3 exclu. Python retourne `[3, 5, 1]`.

`L[1:]` renvoie tous les éléments dont les indices sont compris entre 1 inclus et (indice du dernier élément de la liste + 1) exclu. Python retourne `[5, 1, 8, 10]`.

On peut créer une liste par compréhension :

```
L1=[3*i for i in range(3)]
```

On obtient : `L1=[0, 3, 6]`.

On peut également utiliser une boucle `for` :

```
L1=[]                    # création d'une liste vide
for i in range(3):       # i varie entre 0 inclus et 3 exclu
    L1.append(3*i):      # ajoute la valeur 3*i à la liste L1
```

On obtient : `L1=[0, 3, 6]`.

Attention aux séparateurs dans l'extraction de tranche et dans la fonction `range` :

```
L[start:stop:step]        # mettre deux points entre start et stop
range(start, stop, step)  # mettre une virgule entre start et stop
```

Cours :
On utilise la même technique pour les tuples et les chaînes de caractères.

```
L=(3, 5, 8, -2)          # tuple
L2=L[0:3]                # L2=(3, 5, 8)
c="C'est un mot"         # chaîne de caractères
c2=c[0:3]                # c2="C'e"
```

Python affiche :

```
L1 = [2, 4, 6]
[2, 4]
```

Liste `L1` : *i* varie entre 2 inclus et 7 exclu avec un pas de 2.

`L1[:-1]` extrait les éléments de `L1` : le dernier élément de `L1` est exclu.

Graphiques

2

Exercice 2.1 : Tracé d'une fonction avec matplotlib

On considère les fonctions f_1 et f_2 définies sur [0, 2] par :

$$f_1(x) = \begin{cases} x & \text{pour } 0 \le x < 1 \\ 1 & \text{pour } 1 \le x \le 2 \end{cases} \quad \text{et } f_2(x) = \sin(x) + 0{,}1.$$

Les fonctions suivantes permettent le tracé de fonctions :

```
import matplotlib.pyplot as plt  # module matplotlib.pyplot renommé plt
plt.figure()          # nouvelle fenêtre graphique
plt.plot(x, y, color='r', linewidth=3, marker='o')
   # color : choix de la couleur
   # ('r' : red, 'g' : green, 'b' : blue, 'black' : black)
   # linewidth : épaisseur du trait
   # marker : différents symboles '+', '.', 'o', 'v'
   # linestyle : style de la ligne ('-' ligne continue,
   # '--' ligne discontinue, ':' ligne pointillée)
plt.plot(x, y, '*')   # points non reliés représentés par '*'
plt.grid()            # affichage de la grille
plt.title('Titre')    # ajout d'un titre
plt.xlabel('axe x')   # affiche 'axe x' en abscisse d'un graphique
plt.ylabel('axe y')   # affiche 'axe y' en ordonnée d'un graphique
plt.axis ([xmin, xmax, ymin, ymax])    # précise les bornes pour les
                                       # abscisses et les ordonnées
plt.legend(['courbe 1', 'courbe 2'])   # permet de légender les courbes
plt.show()            # affiche la figure à l'écran
```

Définir les deux fonctions `f1` et `f2` dans Python en utilisant le module `math`. Tracer les courbes représentatives des deux fonctions sur l'intervalle [0, 2] avec un pas de 0,05. Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de listes.
- Courbe représentative de `f1` : épaisseur du trait égale à 3, couleur bleue.
- Courbe représentative de `f2` : points non reliés représentés par *, couleur rouge.
- Légender les deux courbes.
- Afficher `'x'` pour l'axe des abscisses et `'y'` pour l'axe des ordonnées.
- Afficher le titre : `'Tracé de fonctions'`.
- Axe des x compris entre 0 et 2, axe des y compris entre 0 et 1,5.