

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

MPSI·PCSI·PTSI
MP·PC·PSI·PT·TSI·TPC

EXERCICES
INCONTOURNABLES

3^e édition

DUNOD

l'intelligence

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>

© Dunod, 2025

11 rue Paul Bert, 92240 Malakoff

www.dunod.com

ISBN 978-2-10-088172-7

Table des matières

Avant-propos	VII
1. Prise en main de Python	1
1.1 : Assertion, moyenne, variance et écart-type d'une liste de nombres	1
1.2 : Boucle, test, fonction (banque PT 2015)	9
1.3 : Variables locales, variables globales	12
1.4 : Affectation, objet immuable, copie	13
1.5 : Passage par référence pour les listes, effet de bord	18
1.6 : Slicing, extraction de tranche	20
2. Graphiques	22
2.1 : Tracé d'une fonction avec matplotlib	22
2.2 : Tracé d'un histogramme avec matplotlib	25
3. Terminaison, complexité	26
3.1 : Comparaison de deux listes (Mines Ponts 2017)	26
3.2 : Amélioration de la complexité (Mines Ponts 2018)	29
3.3 : Décomposition en base b d'un entier (CCINP MP Maths 2015)	31
3.4 : Recherche du nombre de zéros (banque PT 2015)	35
3.5 : Comparaison de textes (X-ENS 2023)	37
4. Algorithmes	40
4.1 : Recherche du minimum et du maximum d'une liste de nombres, complexité	40
4.2 : Assertion. Recherche du maximum, du second maximum d'une liste	41
4.3 : Recherche d'un mot dans un texte, boucles imbriquées	42
5. Algorithmes de dichotomie	45
5.1 : Recherche d'un élément dans une liste non triée, algorithme naïf, complexité	45
5.2 : Recherche dichotomique dans une liste triée, complexité	46

5.3 : Recherche dichotomique de la bonne orientation d'une image (CCINP PSI-PC 2023)	49
6. Récursivité	54
6.1 : Factorielle d'un entier naturel	54
6.2 : Exponentiation naïve, exponentiation rapide	57
6.3 : Tours de Hanoï	62
6.4 : Recherche du pgcd (CCINP MP Maths 2016)	66
6.5 : Recherche dichotomique dans une liste triée, version récursive	70
6.6 : Dessins de fractales	72
6.7 : Suite des nombres de Fibonacci	77
7. Algorithmes gloutons (sauf TSI et TPC)	79
7.1 : Rendu de monnaie	79
7.2 : Problème du sac à dos	81
7.3 : Allocation de salle de spectacles	82
7.4 : Justification d'un paragraphe, algorithme glouton (Mines Ponts 2023)	84
8. Lecture et écriture de fichiers	86
8.1 : Lecture et écriture de fichiers, calculs statistiques	86
8.2 : Lecture de fichiers	89
9. Matrices de pixels et images, traitement d'images	92
9.1 : Traitement d'images et filtrage passe-bas	92
9.2 : Filtrage d'images	97
9.3 : Typographie informatisée, manipulation de listes de listes (Mines Ponts 2023)	100
9.4 : Rastérisation, affichage de texte (Mines Ponts 2023)	104
9.5 : Rotation d'une image (CCINP PSI-PC 2023)	110
10. Tris	117
10.1 : Tri par insertion	117
10.2 : Tri par sélection	120
10.3 : Tri rapide (sauf TSI et TPC)	123
10.4 : Tri par partition-fusion	126
10.5 : Tri par comptage, histogramme	130
10.6 : Tri à bulles (sauf TSI et TPC)	133

11. Dictionnaire, pile, file, deque	136
11.1 : Opérations de base sur les dictionnaires	136
11.2 : Nombre d'occurrences de chaque élément d'une liste à l'aide d'un dictionnaire	140
11.3 : Nombre d'occurrences de chaque caractère dans une chaîne de caractères (Mines Ponts 2024)	141
11.4 : Opérations de base sur les piles	142
11.5 : Parenthésage	144
11.6 : Comparaison de textes, dictionnaire, pile (X-ENS 2023)	145
11.7 : Opérations de base sur les files	150
11.8 : Utilisation des deque	153
12. Graphes	156
12.1 : Matrice d'adjacence	156
12.2 : Graphe avec liste d'adjacence. Dictionnaire des sommets adjacents	161
12.3 : Graphe avec liste d'adjacence. Liste des sommets adjacents	163
12.4 : Parcours en largeur d'un arbre en utilisant une file	164
12.5 : Parcours en largeur d'un graphe avec une deque	168
12.6 : Parcours en largeur d'un graphe avec une matrice d'adjacence	171
12.7 : Parcours en profondeur d'un graphe	174
12.8 : Algorithme récursif du parcours en largeur	178
12.9 : Algorithme récursif du parcours en profondeur	180
12.10 : Recherche d'un cycle, graphe non orienté, parcours en largeur	182
12.11 : Test de connexité d'un graphe, parcours en largeur, liste d'adjacence	185
12.12 : Flot maximal et coupe minimale (CCINP PSI-PC 2023)	188
12.13 : Test de connexité d'un graphe – Parcours en profondeur – Arbre couvrant	
13. Recherche d'un plus court chemin (sauf TSI et TPC)	197
13.1 : Recherche d'un plus court chemin, graphe orienté	197
13.2 : Algorithme de Dijkstra et liste d'adjacence	199
13.3 : Algorithme de Dijkstra	210
13.4 : Algorithme A*	217
13.5 : Algorithme A* avec liste d'adjacence	222
13.6 : Algorithme A*, distance de Manhattan	228
13.7 : Variantes de l'algorithme A*, distance de Manhattan	

14. Programmation dynamique (Spé) (sauf TSI et TPC)	234
14.1 : Suite des nombres de Fibonacci, Top Down et Bottom Up	234
14.2 : Rendu de monnaie	237
14.3 : Problème du sac à dos	246
14.4 : Algorithme de Floyd-Warshall	254
14.5 : Justification d'un paragraphe, équation de Bellman (Mines Ponts 2023)	260
15. Intelligence artificielle et jeu à deux joueurs (Spé)	264
15.1 : Algorithme des k plus proches voisins	264
15.2 : Matrice de confusion, valeur optimale des k plus proches voisins	269
15.3 : Algorithme des k -moyennes	275
15.4 : Jeu de morpion, algorithme min-max	283
15.5 : Jeu de morpion, algorithme min-max et profondeur	291
15.6 : Classification automatique des caractères (CCINP PSI-PC 2023)	293
16. Bases de données (Spé)	298
16.1 : Joueurs de tennis	298
16.2 : Tournois et joueurs de tennis	302
16.3 : Numéros de sécurité sociale	307
16.4 : Imprimantes (Mines Ponts 2015)	311
16.5 : Paludisme (Mines Ponts 2016)	313
16.6 : Gestion de polices de caractères vectorielles (Mines Ponts 2023)	316
16.7 : Base de données de caractères (CCINP PSI-PC 2023)	318
17. Algorithmique numérique (Spé) (uniquement TSI et TPC)	320
17.1 : Équation différentielle du premier ordre	320
17.2 : Équation différentielle du deuxième ordre	324
17.3 : Résolution d'un système linéaire par la méthode de Gauss	327
17.4 : Interpolation polynomiale de Lagrange	336
17.5 : Interpolation par morceaux	339
Index	343

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 », « Dictionnaire, 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>

Prise en main de Python

1

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` à pour indice 0 : 'p'

`s1[n-1]` : dernier caractère : 'e'

`C1='ab'`

`C2=C1*3` : retourne 'abcabcabcab', 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 deque). 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)

```

• **Deque** : 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 deque »). On peut ajouter et supprimer des éléments aux deux extrémités. Pas d'extraction de tranche (ou slicing) avec les deque.

```

from collections import deque : module permettant d'utiliser les deque
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, deque (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 $X_1, X_2 \dots X_N$. 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 ?