Eléments de Programmation (en Python)
Recueil d'exercices – Saison 1

Sorbonne Université – Licence 1 – 2022/2023

Table des matières

Thème 1 : Exercices numériques simples

Exercice 1.1 : Moyenne de trois nombres (corrigé)

Cet exercice a pour but de trouver les paramètres d'une fonction pour résoudre un problème, et d'écrire la spécification et la définition de fonctions très simples.

Question 1

Donner une définition de la fonction `moyenne_trois_nb` qui effectue la moyenne arithmétique de trois nombres.

Dans le jeu de tests, on vérifiera notamment le calcul de moyenne des nombres 3, 6 et -3 puis de -3, 0 et 3 puis de 1.5, 2.5 et 1.0 (on pourra ajouter d'autres tests en complément).

Question 2 : moyenne pondérée

Écrire une définition de la fonction `moyenne_ponderee` qui effectue la moyenne de trois nombres `a`, `b`, `c` avec des coefficients de pondération, respectivement `pa` (pondération en `a`), `pb` et `pc`.

Proposer un jeu de tests comprenant *au moins* trois tests.

Exercice 1.2 : Calcul d'un prix TTC (corrigé)

Le but de cet exercice est d'effectuer des conversions simples entre des *prix hors-taxe* (HT) et des *prix toutes taxes comprises* (TTC). Ceci permet de s'intéresser au passage d'un problème de la vie courante à une solution informatique.

Question 1

Ecrire une définition de la fonction `prix_ttc` qui calcule le prix toutes taxes comprises (TTC) à partir d'un prix hors taxe (HT) et d'un taux de TVA exprimé en pourcentage, par exemple `20.0` pour une TVA de 20%.

Par exemple :

```
>>> prix_ttc(100.0, 20.0)
120.0
```

```
>>> prix_ttc(100, 0.0)     # remarque : conversion implicite de l'entier 100
                           #            en le flottant équivalent 100.0
100.0
```

```
>>> prix_ttc(100, 100.0)
200.0
```

```
>>> prix_ttc(0, 20)
0.0
```

```
>>> prix_ttc(200, 5.5)
211.0
```

Question 2

Donner une définition de la fonction `prix_ht` qui calcule le prix hors taxe à partir du prix toutes taxes comprises et du taux de TVA.

Remarque : votre jeu de tests doit correspondre à celui proposé pour la fonction `prix_ttc`.

Exercice 1.3 : Calcul de fonctions polynômiales (corrigé)

Cet exercice a pour but de définir des fonctions de calcul de polynômes. On souhaite mettre l'accent sur l'efficacité des algorithmes utilisés (minimisation du nombre de multiplications).

Question 1

Après avoir spécifié le problème, écrire un jeu de tests et donner une définition de la fonction `polynomiale` telle que `polynomiale(a, b, c, d, x)` évalue le polynôme $ax^3 + bx^2 + cx + d$ (à coefficients flottants).

Par exemple :

```
>>> polynomiale(1, 1, 1, 1, 2)
15
```

```
>>> polynomiale(1, 1, 1, 1, 3)
40
```

Quel est le nombre de multiplications effectuées par votre définition ? Peut-on faire mieux ? Si oui, proposer une version plus efficace, sinon justifiez.

Question 2

Après avoir spécifié le problème, écrire un jeu de tests et donner une définition de la fonction `polynomiale_carre` qui rend la valeur de $ax^4 + bx^2 + c$.

Par exemple :

```
>>> polynomiale_carre(1, 1, 1, 2)
21
```

```
>>> polynomiale_carre(1, 1, 1, 3)
91
```

Quel est le nombre de multiplications effectuées par votre définition ? Pouvez-vous proposer une version plus efficace ?

Exercice 1.4 : Aire d'une couronne

Cet exercice a pour but de calculer l'aire d'une couronne (c'est-à-dire l'aire comprise entre 2 disques de même centre mais de rayons différents), et de travailler sur la notion d'hypothèse.

Question 1

Donner une définition ainsi qu'un jeu de tests de la fonction `aire_disque` qui calcule l'aire πr^2 d'un disque de rayon r

Remarque : En python, la constante π est déjà définie dans le module `math`. Pour l'utiliser, il faut *déclarer* l'utilisation de ce module en tête du programme avec l'instruction suivante :

```
import math
```

Ensuite, la constante peut être utilisée :

```
>>> math.pi
3.141592653589793
```

Question 2

Donner une définition ainsi qu'un jeu de tests de la fonction `aire_couronne` qui, étant donné deux nombres r_1 et r_2, calcule l'aire de la couronne de rayon intérieur r_1 et de rayon extérieur r_2.

Par hypothèse, on considère que le rayon intérieur est inférieur ou égal au rayon extérieur.

Exercice 1.5 : Conversion Degrés Fahrenheit-Celsius

Cet exercice a pour but de définir des fonctions de conversion de températures données en degrés Celsius en leur équivalent en degrés Fahrenheit et réciproquement.

On rappelle que pour mesurer une température, on utilise en France l'échelle des degrés Celsius alors que, aux USA par exemple, c'est l'échelle des degrés Fahrenheit qui est utilisée.

Question 1

Écrire une définition de la fonction `fahrenheit_vers_celsius` qui convertit une température `t` exprimée en degrés *Fahrenheit* en son équivalent en degrés *Celsius*.

Rappel : la température t en degrés Fahrenheit équivaut à la température $(t - 32) * \frac{5}{9}$ en degrés Celsius.

Par exemple :

```
>>> fahrenheit_vers_celsius(212)
100.0
```

```
>>> fahrenheit_vers_celsius(32)
0.0
```

```
>>> fahrenheit_vers_celsius(41)
5.0
```

Question 2

Donner une définition de la fonction `celsius_vers_fahrenheit` qui effectue la conversion inverse.

Exercice 1.6 : Calcul du n-ième nombre de Fermat

Les nombres de Fermat sont, en arithmétique, des nombres qui s'écrivent sous la forme $F_n = 2^{2^n} + 1$ *pour* n *positif. Fermat conjectura qu'ils étaient tous premiers (ils sont en fait, premiers* *jusqu'à F_4, et non-premiers de F_5 à F_{32}. On connaît peu la primalité de ces nombres à partir de F_{33}).*

Le but de cet exercice est d'écrire un programme permettant de calculer, étant donné une valeur n quelconque, le n-ième nombre de Fermat.

Question 1

Donner une définition et un jeu de tests de la fonction `fermat` qui calcule F_n, le n-ième nombre de Fermat dont la valeur est $2^{2^n} + 1$.

Rappel : La fonction exponentiation est définie dans la carte de référence.

Par exemple :

```
>>> fermat(0)
3
```

```
>>> fermat(1)
5
```

```
>>> fermat(2)
17
```

```
>>> fermat(5)
4294967297
```

Question 2

En Python, proposer une expression permettant de vérifier que F_5 est divisible par 641, et n'est donc pas premier.

Exercice 1.7 : Coût d'une excursion

Le but de cet exercice est de réfléchir à la division entière en écrivant un programme implémentant une formule qui fait usage de cette notion de manière non triviale.

Question 1

Une association propose des excursions à la journée. Ses frais incluent :

— le transport en autocars de 60 places, facturé 1200 euros la journée par autocar.
— le salaire de guides touristiques, à raison d'un guide pour 18 personnes maximum, facturé 300 euros la journée par guide.

Donner la définition et un jeu de tests de la fonction `excursion` qui étant donné un entier `nb_pers` calcule le coût (minimum) pour l'association d'une excursion de `nb_pers` personnes.

Remarque: Tout autocar contenant au moins une personne (de même, tout guide affecté à au moins une personne) doit être payé en totalité.

Par exemple :

```
>>> excursion(0) # aucune personne : on doit payer 0 autocar et 0 guide
0
```

```
>>> excursion(1) # 1 personne : on doit payer 1 autocar et 1 guide
1500
```

```
>>> excursion(18) # 18 personnes : on doit payer 1 autocar et 1 guide
1500
```

```
>>> excursion(60) # 60 personnes : on doit payer 1 autocar et 4 guides
2400
```

```
>>> excursion(61) # 61 personnes : on doit payer 2 autocars et 4 guides
3600
```

```
>>> excursion(150) # 150 personnes : on doit payer 3 autocars et 9 guides
6300
```

Question 2

La même association propose maintenant une nouvelle excursion pour adultes et enfants. Ses frais incluent :

— le transport en autocars de 60 places (adultes et enfants), facturé 1200 euros la journée par autocar.
— le salaire de guides pour adultes, à raison d'un guide pour 18 adultes maximum, facturé 300 euros la journée par guide.
— le salaire d'animateurs pour enfants, à raison d'un guide pour 8 enfants maximum, facturé 250 euros la journée par animateur.

Donner la définition et un jeu de tests de la fonction `excursion2` qui prend comme entrée deux entiers `nb_adu` et `nb_enf` et calcule le coût (minimum) pour l'association d'une excursion avec `nb_adu` adultes et `nb_enf` enfants. Par exemple:

```
>>> excursion2(0,0) # ni adulte, ni enfant : 0 car, 0 guide, 0 animateur
0
```

```
>>> excursion2(1,0) # 1 adulte, pas d'enfant : 1 car, 1 guide, 0 animateur
1500
```

```
>>> excursion2(0,1) # pas d'adulte, 1 enfant : 1 car, 0 guide, 1 animateur
1450
```

```
>>> excursion2(18,0) # 18 adultes, pas d'enfant : 1 car, 1 guide, 0 animateur
1500
```

```
>>> excursion2(0,8) # pas d'adulte, 8 enfants : 1 car, 0 guide, 1 animateur
1450
```

```
>>> excursion2(18,8) # 18 adultes, 8 enfants : 1 car, 1 guide, 1 animateur
1750
```

```
>>> excursion2(36,14) # 36 adultes, 14 enfants : 1 car, 2 guides, 2 animateurs
2300
```

```
>>> excursion2(150,120) # 150 adultes, 120 enfants : 5 cars, 9 guides, 15 anim.
12450
```

Exercice 1.8 : Prise en main de l'environnement MrPython (sur machine)

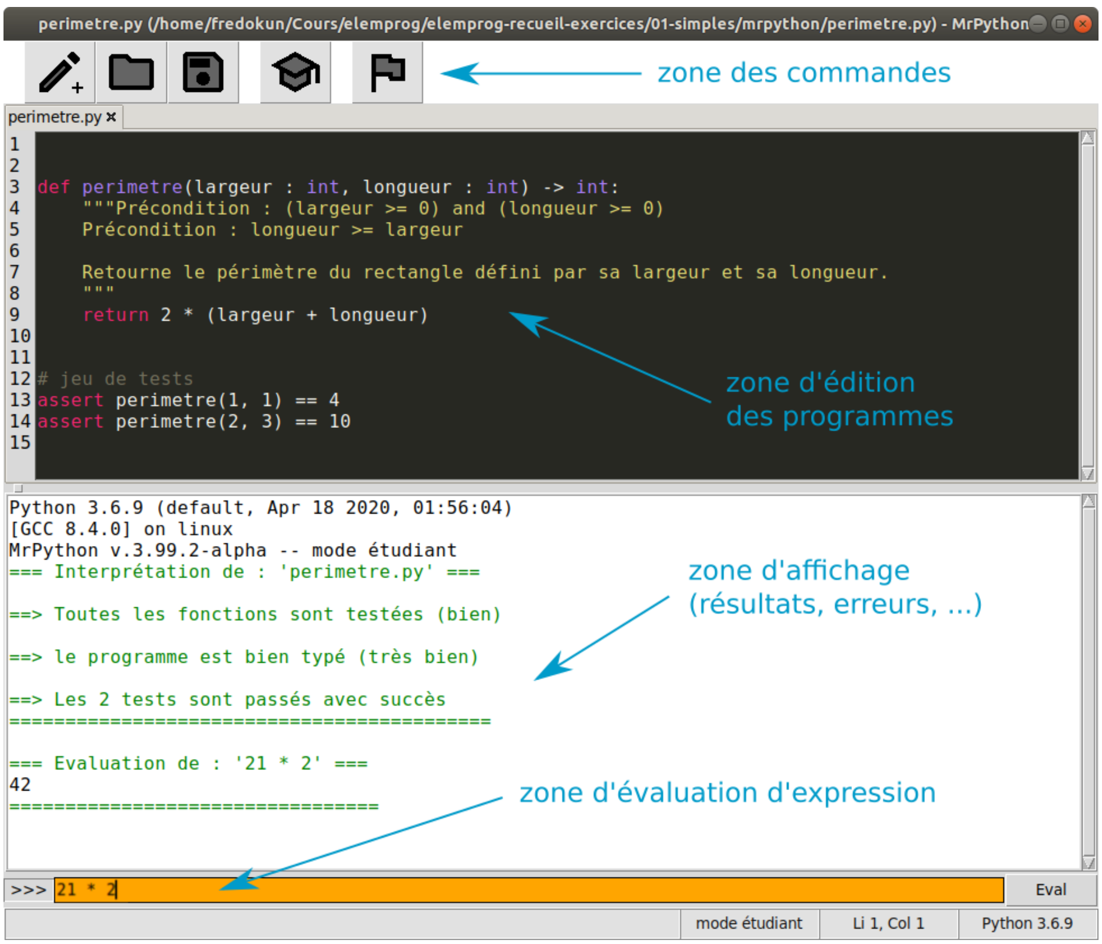
MrPython est un environnement graphique pour la programmation en Python. Il a été développé entièrement en Python avec la bibliothèque graphique tkinter.

MrPython est un environnement à vocation principalement pédagogique, il propose notamment un support spécifique pour le cours : le *mode étudiant*. Il est également possible de programmer en *mode expert* qui permet de programmer sans restriction en Python.

En salle machine, et donc dans ce TME, nous n'utiliserons que le mode étudiant.

Présentation de MrPython

La figure ci-dessous présente une session typique d'utilisation de MrPython.



L'environnement propose trois zones principales :

— la **zone de commandes** qui consiste en un nombre réduit de commandes
  — créer, charger et sauvergarder un fichier Python
  — alterner entre le mode *étudiant* et le mode *expert*
  — lancer l'interprétation du programme courant

- la **zone d'édition** qui permet de rédiger un programme Python de façon confortable
  - coloration syntaxique
  - gestion «intelligente» de l'indentation
  - commandes usuelles d'édition (Ctrl-C, Ctrl-V, etc.)
- la **zone d'interaction** composée de deux «sous-zones» :
  - la **zone d'affichage** qui affiche les résultats d'évaluation, les erreurs éventuelles, etc.
  - la **zone d'évaluation** permettant de saisir une expression Python simple (sur une ligne) et de l'évaluer (bouton `Eval` ou simplement retour chariot – touche *Entrée*).

Premier pas en Python avec MrPython

Pour lancer l'application `mrpython` (en salle machine en environnement Linux)

1. ouvrir le menu des *Applications* et chercher l'application «Terminal» pour ouvrir une fenêtre de console (*shell*) vous permettant de taper des commandes Linux et de les exécuter.
2. taper la commande mrpython suivi de la touche «Entrée»

Au lancement de MrPython la zone d'édition est vide et trois actions principales sont disponibles:

- créer un nouveau programme Python
- charger un programme Python depuis un fichier `.py`
- saisir des expressions Python dans la zone d'évaluation.

Pour ce premier exercice nous nous contenterons d'évaluer quelques expressions simples.

Question 1

À la suite de l'invite `>>>`, tapez dans la zone d'évaluation les expressions suivantes et étudiez leur évaluation :

```
42
```

```
type(42)
```

```
2.3
```

```
type(2.3)
```

```
True
```

```
type(True)
```

```
"chaine de caractères"
```

```
type("chaine de caractères")
```

```
3+2
```

```
6*2
```

```
6**2
```

```
10 > 23
```

Question 2

Tapez successivement dans la zone d'évaluation les lignes suivantes :

```
"essai" == "essai"
```

```
"essai" == "essai "
```

```
"3" == 3
```

```
4 == 5
```

```
4 == 4
```

```
4 == 2*2
```

Expliquez chacun des résultats obtenus.

Question 3

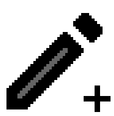
Écrivez les expressions qui permettent de :

- calculer la division (réelle) de 42 par 5
- calculer le quotient de la division euclidienne de 42 par 5
- calculer le reste de la division euclidienne de 42 par 5
- tester l'égalité entre 6 et $2 * 3$
- calculer la valeur de l'expression arithmétique $\frac{2*3^3}{5-2}$
- tester le type de la valeur calculée précédemment (est-ce le résultat que vous attendiez ?)
- calculer le maximum entre cette valeur et le nombre 10 (se servir de la carte de référence)

Question 4

Pour programmer en Python, nous ne pouvons nous contenter de saisir des expressions dans la zone d'évaluation. Pour écrire un programme Python nous devons exploiter la **zone d'édition**.

1) Commencez par créer un nouveau programme. Pour cela, il suffit de cliquer sur l'icône **Nouveau fichier**



(on peut également utiliser la combinaison de touches: `Ctrl-N`).

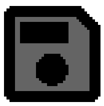
Une fois la zone d'édition active, recopiez la fonction `perimetre` vue en cours :

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (largeur >= 0) and (longueur >= 0)
    Précondition : longueur >= largeur

    Retourne le périmètre du rectangle défini par sa largeur et sa longueur.
    """
    return 2 * (largeur + longueur)
```

Remarque : il est aussi possible d'ouvrir un fichier existant avec la commande **Ouvrir fichier**

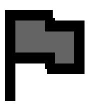
2) Enregistrez le programme dans un fichier `perimetre.py` en cliquant sur l'icône **Sauvergarder**



(on peut également utiliser la combinaison de touches: `Ctrl-S`).

Cette étape est importante et réclamée avant toute exécution du programme. **ATTENTION : Pensez à sauvegarder régulièrement vos programmes.** (le plus simple est de régulièrement appuyer sur `Ctrl-S`).

3) Exécutez votre programme en cliquant sur l'icône **Exécution**

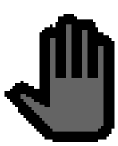


(on peut également utiliser la combinaison de touches: `Ctrl-R`).

Si aucune erreur ne s'est produite, alors on peut désormais utiliser la fonction `perimetre` depuis la zone d'évaluation. À la suite de l'invite, écrivez une expression qui calcule le périmètre d'un rectangle de largeur 4 unités et de longueur 5 unités.

Ajoutez ensuite un petit jeu de tests après la fonction dans la zone d'édition et relancez l'éxécution du programme pour vérifier que vos tests passent bien.

Remarque : il est possible qu'une exécution ne se termine pas (en raison des boucles que nous aborderons un peu plus tard) ou plus simplement prenne trop de temps. Dans ce cas, il est possible de cliquer sur l'icône **Stopper**, ce qui interrompt l'exécution du programme.



Question 5

A la suite de la fonction précédente, dans la fenêtre d'édition, rajoutez la définition de la fonction `surface` qui calcule la surface d'un rectangle de largeur `larg` et longueur `long`. Vous n'oublierez pas de fournir un jeu de tests pour votre fonction.

Vérifiez ensuite le résultat de ce que renvoie votre fonction sur les exemples suivants :

```
>>> surface(1,4)
```

```
>>> surface(2,0)
```

```
>>> surface(3,4)
```

Attention : n'oubliez pas de sauvegarder votre fichier avant d'exécuter le programme.

Exercice 1.9 : Dessins avec le module gfx (sur machine)

L'environnement MrPython intègre en mode étudiant une petite bibliothèque graphique `studentlib.gfx` que nous allons découvrir dans cet exercice.

En règle générale, les bibliothèques Python sont composées de *modules* devant être importés avec la commande `import` (comme `import math` pour la bibliothèque mathématique). En mode étudiant, MrPython réalise 'importation de la bibliothèque graphique automatiquement. En d'autres termes il n'y a rien à faire pour utiliser cette bibliothèque (sauf, bien sûr, de basculer sur le mode étudiant si nécessaire).

Fonctionalités de la bibliothèque gfx

La bibliothèque gfx a été conçue avec un objectif de concision et de simplicité. Elle introduit un nouveau type `Image` et trois catégories de fonction:

1. les images primitives: ligne, triangle (contour ou plein) et ellipse (coutour ou plein),
2. les combinateurs d'images: superposition et superposition inverse,
3. la fonction d'affichage `show_image`.

Toutes les fonctions opèrent dans un repère cartésien centré sur la coordonnée $(0, 0)$. Le point en bas à gauche du repère est $(-1, -1)$ et le point en haut à droite est $(1, 1)$. Les figures peuvent être colorées, avec la couleur décrite par une chaîne de caractères (type `str`). Lorsque la couleur (le dernier argument dans les fonctions d'images primitives) n'est pas précisée, le noir (chaîne `"black"`) est choisi par défaut. D'autres choix possibles sont: `"red"` `"green"` `"blue"` `"yellow"` `"purple"`...

Les spécifications de ces fonctions sont données ci-dessous.

Remarque: Il ne faut, évidemment, pas réécrire ces fonctions dans la zone d'édition ; elles sont déjà définies dans MrPython, leur spécification est donnée en vue de leur utilisation.

Images primitives

```
def draw_line(x0 : float, y0 : float
              , x1 : float, y1 : float, color :str ="black") -> Image:
    """Construit une image représentant un segment de droite
    entre les points (x0,y0) et (x1,y1).

    La couleur (argument optionel, ligne noire par défaut)
    peut être spécifiée par une chaîne de caractères."""
```

```python
def draw_triangle(x0 : float, y0 : float
                 , x1 : float, y1 : float
                 , x2 : float, y2 : float, color : str = "black") -> Image:
    """Construit une image représentant le contour d'un triangle reliant les
    trois points (x0,y0) et (x1,y1) et (x2, y2)."""
```

```python
def fill_triangle(x0 : float, y0 : float
                 , x1 : float, y1 : float
                 , x2 : float, y2 : float, color : str = "black") -> Image:
    """Construit une image représentant un triangle plein reliant les
    trois points (x0,y0) et (x1,y1) et (x2, y2)."""
```

```python
def draw_ellipse(x0 : float, y0 : float
                , x1 : float, y1 : float, color : str = "black"):
    """Construit une image représentant le contour d'une ellipse
    inscrite dans le rectangle dont le point en haut à gauche est (x0,y0)
    et le point en bas à droite est (x1,y1)."""
```

```python
def fill_ellipse(x0 : float, y0 : float
                , x1 : float, y1 : float, color : str = "black"):
    """ float * float * float * float (* str) -> Image

    Construit une image représentant une ellipse pleine
    inscrite dans le rectangle dont le point en haut à gauche est (x0,y0)
    et le point en bas à droite est (x1,y1)."""
```

Combinateurs d'images

```python
def overlay(image1 : Image, image2 : Image, ...):
    """Construit l'image résultat de la superposition des images
    passées en argument. La deuxième image se place *au-dessus* de la
    première, la troisième *au-dessus* de la seconde, etc.
```

```python
def underlay(image1 : Image, image2 : Image, ...):
    """Construit l'image résultat de la superposition des images
    passées en argument. La deuxième image se place *au-dessous* de la
    première, la troisième *au-dessous* de la seconde, etc.
```

Fonction d'affichage

Les images construites par les fonctions ci-dessus ne sont pas affichées directement. Ceci permet notamment de construire des images complexes, résultant de nombreuses superpositions, sans afficher toutes les étapes intermédiaires de la construction. La fonction ci-dessous permet d'afficher une image dans une fenêtre dite de *canevas*.

```python
def show_image(image : Image) -> None:
    """Réalise l'affichage de l'image passée en argument
    dans une fenêtre de canevas."""
```

Question 1

Construire l'image d'un segment de droite entre les points de coordonnées $(-0.5, 0.2)$ et $(0.7, -0.5)$. Pouvez-vous dessiner (en gros) sur papier l'image résultante ? Vérifier en lançant l'affichage de cette image (avec `show_image`).

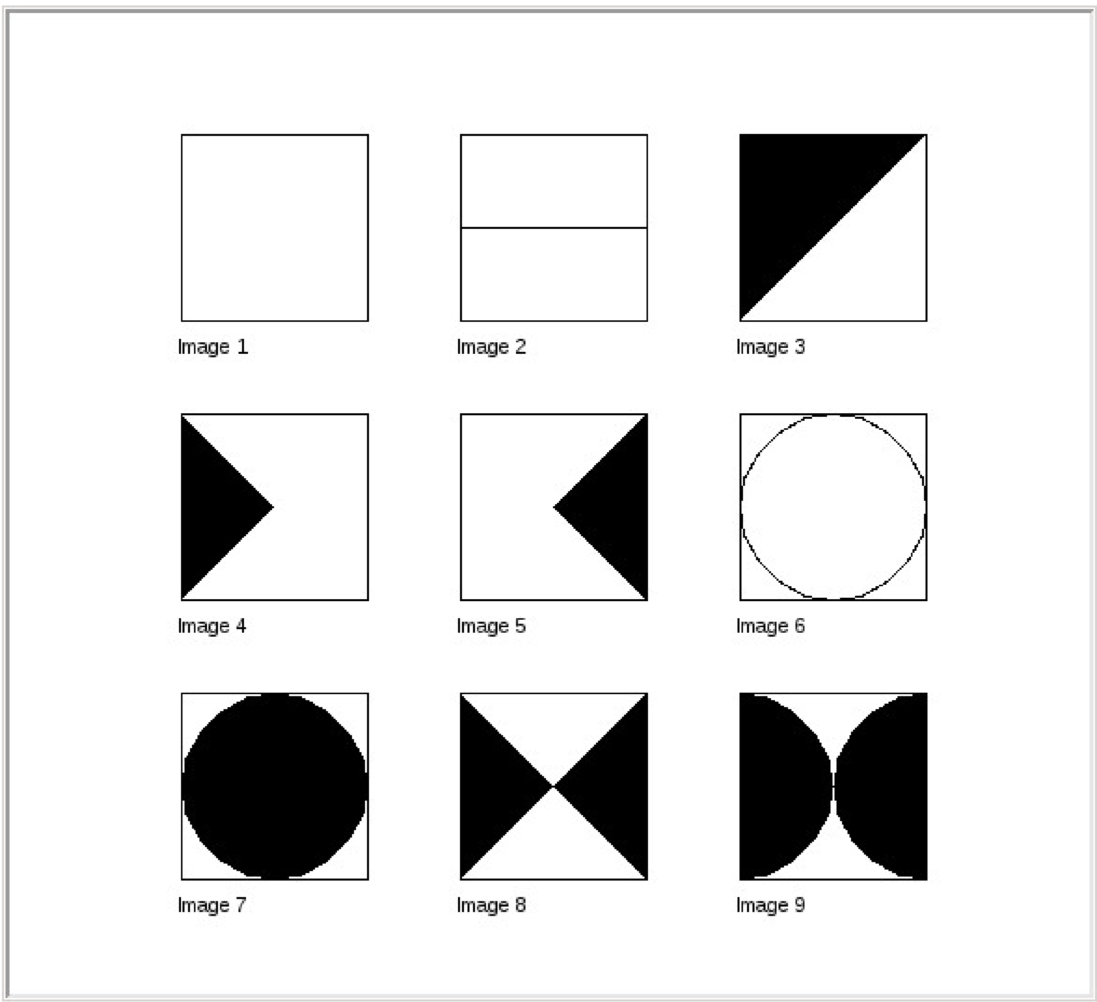
Question 2

Définir une fonction `dessine_carre` prenant trois arguments `x`, `y`,`c` et construisant l'image du carré *plein* dont le point en bas à gauche est de coordonnées `(x,y)` est le côté et de longueur `c`.

Utiliser cette fonction pour construire (et afficher) l'image du carré de côté 1 centré sur l'origine.

Question 3

Construire (et afficher) les images ci-dessous :



Thème 2 : Exercices avec fonctions simples

Exercice 2.1 : Variables et affectations (corrigé)

Question 1

On considère la suite d'instructions (affectations de variables et expressions simples) ci-dessous.

Après chaque ligne :

— s'il s'agit d'une affectation donner la table des variables (nom / valeur) après interprétation de cette instruction, et
— s'il s'agit d'une initialisation, préciser la déclaration de variable correspondante.
— s'il s'agit d'une expression donner le type de l'expression ainsi que sa valeur.

```
a = 3
```

```
a
```

```
a = 0
```

```
a
```

```
a == 0
```

```
a == 3
```

```
b = 2
```

```
a == b
```

```
b = a
```

```
b
```

```
a == b
```

```
b = 4
```

```
a == b
```

```
(a == 2) == (b == 0)
```

```
c = (a == 0)
```

```
c
```

```
c == (b == 3)
```

```
x
```

Question 2

D'après-vous, que se passe-t-il si l'on soumet la définition suivante à l'interprète Python (à essayer ensuite en TME) ?

```
def sante(x : float) -> str:
    """Retourne "Bonne santé" si x vaut 37.5, retourne "Malade" sinon.
    """
```

```python
    if x = 37.5:
        return "Bonne santé"
    else:
        return "Malade"
```

Exercice 2.2 : Calcul des mentions (corrigé)

Cet exercice a pour but de faire manipuler des *alternatives* multiples et/ou imbriquées. Il s'agit de calculer la mention correspondant à une note sur 20.

Question 1

Donner une définition en Python de la fonction `mention` qui calcule la mention correspondant à une note (sur 20) donnée en utilisant les intervalles de notes suivants :

— $[0, 10[\rightarrow$ Eliminé,
— $[10, 12[\rightarrow$ Passable,
— $[12, 14[\rightarrow$ AB,
— $[14, 16[\rightarrow$ B,
— $[16, 20] \rightarrow$ TB.

Les mentions seront représentées par des chaînes de caractères encadrées par des guillemets simples. La mention Eliminé sera par exemple représentée par la valeur `'Eliminé'` de type `str`.

Voici quelques exemples d'applications de la fonction `mention` :

```python
>>> mention(0)
'Eliminé'
```

```python
>>> mention(8)
'Eliminé'
```

```python
>>> mention(10)
'Passable'
```

```python
>>> mention(12.5)
'AB'
```

```python
>>> mention(15)
'B'
```

```python
>>> mention(20)
'TB'
```

Remarque : penser à utiliser le mot-clef `elif` pour les alternatives multiples.

Question 2

Définir à nouveau la fonction `mention`, cette fois-ci en imbriquant les structures conditionnelles.

L'objectif est de minimiser le nombre maximum de tests conditionnels effectués, d'une part, et de minimiser le nombre moyen de tests, d'autre part, en fonction de la répartition des notes, en supposant que la majorité des notes se situent entre 10 et 12.

Exercice 2.3 : Couverture (corrigé)

On cherche dans cet exercice à définir un jeu de tests permettant d'explorer tous les cas de tests d'une fonction.

Question 1

La fonction suivante rend 6 valeurs différentes (une chaîne de caractères) selon l'ordre dans lequel sont donnés ses trois arguments, **différents deux à deux**.

```
def f(n1 : float, n2 : float, n3 : float) -> str:
    """Précondition : n1 != n2 and n2 != n3 and n3 != n1
    retourne un cas parmi 6 selon les valeurs de n1, n2 et n3.
    """
    if n1 < n2 and n2 < n3:
        return 'cas 1'
    elif n1 < n3 and n3 < n2:
        return 'cas 2'
    elif n2 < n1 and n1 < n3:
        return 'cas 3'
    elif n2 < n3 and n3 < n1:
        return 'cas 4'
    elif n3 < n1 and n1 < n2:
        return 'cas 5'
    else:
        return 'cas 6'
```

Définir un jeu de six tests vérifiant les six cas possibles.

Question 2

Donner une autre définition de `f` sans utiliser les opérateurs logiques `and` et `or` mais uniquement des alternatives. Vérifier que les résultats obtenus sont identiques à ceux fournis dans le jeu de tests de la question précédente.

Exercice 2.4 : Mesures fiables (corrigé)

Dans cet exercice, nous explorons le problème de l'égalité entre flottants qui est le plus souvent considérée *à epsilon près*. Ceci donne l'occasion de réfléchir à la complétude des jeux de tests. De plus, les alternatives multiples sont ici d'une grande utilité.

Question 1

Donner la définition du prédicat `egal_eps` qui teste l'égalité de deux nombres `x1` et `x2` *à epsilon près*, c'est-à-dire si la valeur absolue de leur différence est inférieure à un nombre strictement positif (et supposé petit) `epsilon` également passé en paramètre.

Le jeu de tests proposé doit être le plus complet possible. Il faut notamment tenir compte du maximum de cas de figures possibles : arguments positifs ou négatifs, rapport entre les deux arguments (plus petit ou plus grand), valeur attendue vraie ou fausse.

Remarque : il est possible soit d'utiliser la fonction prédéfinie `abs` pour le calcul de la valeur absolue, soit de redéfinir la fonction `valeur_absolue` vue en cours.

Question 2

Lors de l'utilisation d'un instrument de mesure (par exemple un thermomètre), il est bon de ne pas se fier à une seule mesure. Soit un instrument de mesure qui dispose de trois capteurs fournissant trois valeurs numériques `v1`, `v2` et `v3` censées donner chacune la mesure d'un même phénomène (par exemple une température). On tolère une différence jugée négligeable entre ces valeurs : en d'autres termes, on considère leur égalité à *epsilon près*.

Attention : l'égalité à *epsilon* près n'est pas transitive en général, elle reste en revanche symétrique.

On cherche à déterminer un taux de fiabilité de la mesure en appliquant le principe suivant: - si les trois valeurs sont deux à deux égales à *epsilon* près, le taux est de $\frac{3}{3}$, c'est-à-dire 1; - si deux couples de valeurs seulement sont égales à *epsilon* près, le taux de fiabilité est de $\frac{2}{3}$ (par exemple `v1` $\approx$ `v3` et `v2` $\approx$ `v3`). - sinon le taux de fiabilité est de $\frac{0}{3}$, c'est-à-dire nul (0).

- Combien y a-t-il de façons d'obtenir le taux de fiabilité de 1, de $\frac{2}{3}$, et de 0 ? Donner des exemples.
- Donner une définition de la fonction `fiabilite` qui donne le taux de fiabilité de trois valeurs `v1`, `v2` et `v3` à `epsilon` près. On prendra soin de bien vérifier tous les cas possibles dans le jeu de tests.

Exercice 2.5 : Volume d'un tétraèdre

Cet exercice a pour but de traduire en Python une formule mathématique non-triviale.

Question 1

Leonhard Euler, célèbre savant du XVIIIème siècle a trouvé une formule concise pour calculer le volume d'un tétraèdre lorsque seules les longueurs de ses six côtés sont connues.

Soient a, b, c, d, e et f les longueurs des six côtés d'un tétraèdre.

La volume de ce tétraèdre est $V = \frac{1}{12}\sqrt{p - q + r}$ avec :

- $p = 4\,.a^2.b^2.c^2$
- $q = a^2.y^2 + b^2.z^2 + c^2.x^2$
- $r = x.y.z$

et :

- $x = a^2 + b^2 - d^2$
- $y = b^2 + c^2 - e^2$
- $z = a^2 + c^2 - f^2$

Proposer une définition de la fonction `volume_tetraedre` qui effectue le calcul décrit ci-dessus.

Par exemple :

```
volume_tetraedre(1, 1, 1, 1, 1, 1)
0.11785113019775792
```

On remarque en passant :

```
>>> import math
>>> math.sqrt(2) / 12
0.11785113019775793
```

Il y a donc un rapport entre cette constante $\frac{\sqrt{2}}{12}$ et le volume d'un tétraèdre.

```
volume_tetraedre(2, 2, 2, 2, 2, 2)
0.9428090415820634
```

Remarque : la constante évoquée précédemment intervient ici aussi.

```
>>> math.sqrt(2) / 12 * (2 * 2 * 2)
0.9428090415820635
```

Question subsidiaire : pour enrichir le jeu de tests, trouver un tétraèdre non-régulier (c'est-à-dire dont les côtés sont de longueurs différentes) dont le volume est calculable.

Il faut savoir en effet que toutes les longueurs ne sont pas possibles pour les tétraèdres. De plus, les contraintes sur les côtés `a`, `b`, ... , `f` ne sont pas triviales. Le plus simple est donc sans doute de dessiner un tétraèdre et de calculer son volume ensuite.

Question 2

Un tétraèdre régulier est tel que tous ses côtés sont de longueurs égales.

Proposer une définition de la fonction `volume_tetraedre_regulier` qui, étant donné une longueur `l`, calcule le volume d'un tel tétraèdre.

Par exemple (comparer avec les exemples précédents) :

```
>>> volume_tetraedre_regulier(1)
0.11785113019775793
```

```
>>> volume_tetraedre_regulier(2)
0.9428090415820635
```

Exercice 2.6 : Manipulation des booléens

Cet exercice a pour but de définir des fonctions sur les booléens en utilisant des alternatives.

Question 1

Sans utiliser les opérateurs logiques `or`*,* `and` *et* `not`, écrire les trois prédicats `ou`, `et` et `non` dont les spécifications sont les suivantes :

```
def ou(p : bool, q : bool) -> bool:
    """Retourne la disjonction de p et q."""

def et(p : bool, q : bool) -> bool:
    """Retourne la conjonction de p et q."""

def non(p : bool) -> bool:
    """Retourne la négation de p."""
```

Par exemple :

```
>>> ou(True, False)
True
```

```
>>> ou(et(True, False), False)
False
```

```
>>> et(ou(False, True), non(False))
True
```

```
>>> non(non(3 == 1 + 2))
True
```

Remarque : les jeux de tests proposés devront couvrir tous les cas possibles.

Question 2

Que se passe-t-il si on évalue l'expression suivante ?

```
ou(3 == 3, 5 // 0 == 2)
```

Quelle est la différence avec l'évaluation de l'expression suivante ?

```
(3 == 3)  or  (5 // 0 == 2)
```

Même questions pour l'expression :

```
et(3 == 4, 5 // 0 == 2)
```

comparée à :

```
(3 == 4) and (5 // 0 == 2)
```

Question 3

En utilisant les fonctions écrites à la question 1, écrire les prédicats `implique` et `ou_exclusif` dont les spécifications sont les suivantes :

```
def implique(p : bool, q : bool) -> bool:
    """Retourne le résultat de 'p implique q'."""

def ou_exclusif(p : bool, q : bool) -> bool:
    """Retourne le résultat de 'p xor q'."""
```

Il peut être utile de se rappeler les formules de logique suivantes :

- `p implique q` est équivalent à `(la négation de p) ou q`.
- `p xor q` est équivalent à `(p et (la négation de q)) ou ((la négation de p) et q)`.

Par exemple :

```
>>> implique(False, False)
True
```

```
>>> implique(True, False)
False
```

```
>>> implique(True, 3 == 3)
True
```

```
>>> ou_exclusif(True, False)
True
```

```
>>> ou_exclusif(3 == 2, 3 == 3)
True
```

```
>>> ou_exclusif(2 == 2, 3 == 3)
False
```

Remarque : encore une fois, on s'assurera que les jeux de tests couvrent tous les cas possibles.

Question 4

En utilisant les prédicats de la question précédente, écrire le prédicat `equivalent` de spécification :

```
def equivalent(p : bool, q : bool) -> bool:
    """Retourne True si et seulement si p et q sont équivalents."""
```

Il peut être utile de se rappeller la formule de logique suivante :

— `p équivaut à q` si et seulement `p implique q` et `q implique p`.

Par exemple :

```
>>> equivalent(True, 3 == 3)
True
```

```
>>> equivalent(True, 3 == 4)
False
```

```
>>> equivalent(3 == 2, 3 == 8)
True
```

Remarque : le jeu de tests de `equivalent` devra encore une fois couvrir tous les cas possibles.

Exercice 2.7 : Dessin d'une tour

L'objectif de cet exercice est de travailler sur les paramètres nécessaires à un problème : bien sûr, il faut qu'il y ait tous les paramètres nécessaires à la définition du problème, mais il ne faut pas qu'il y en ait en plus. Autrement dit les différents paramètres doivent être indépendants. Encore autrement dit, on doit pouvoir appeler une telle fonction avec n'importe quelles valeurs respectant les hypothèses.

Dans cet exercice, les fonctions à écrire utilisent la bibliothèque graphique `gfx`.

Question 1

Nous voudrions définir la fonction `tour` qui construit l'image d'une tour à deux étages comme celles données ci-dessus. Les deux étages d'une même tour ont la même hauteur et chaque tour présente une symétrie verticale.

Figure 1 – Tours

Quels sont les paramètres nécessaires ?

En déduire une spécification de la fonction `tour`

Question 2

Donner une définition de la fonction `rectangle` permettant de dessiner un rectangle peint en noir à partir de son coin bas-gauche `(x,y)` sa longueur `l` et sa hauteur `h`.

Question 3

Donner une définition de la fonction `tour` correspondant aux paramètres de la question 1.

Thème 3 : Exercices simples sur les boucles

Exercice 3.1 : Somme des impairs (corrigé)

Question 1

Donner une définition de la fonction `somme_impairs_inf` telle que `somme_impairs_inf(n)` renvoie la somme des entiers impairs inférieurs ou égaux à `n`.

Par exemple :

```
>>> somme_impairs_inf(1)
1
```

```
>>> somme_impairs_inf(2)
1
```

```
>>> somme_impairs_inf(5)
9
```

Question 2

Donner une définition de la fonction `somme_premiers_impairs` telle que `somme_premiers_impairs(n)` renvoie la somme des `n` premiers entiers impairs.

Par exemple :

```
>>> somme_premiers_impairs(1)
1
```

```
>>> somme_premiers_impairs(2)
4
```

```
>>> somme_premiers_impairs(5)
25
```

Remarque : comme les exemples ci-dessus le suggérent, la somme des n premiers impairs vaut n^2 (le démontrer est un bon exercice mathématique). On exploitera cette propriété dans les jeux de tests (rappel : l'opérateur d'élévation à la puissance est `**` en Python).

Question 3

Effectuer la simulation de boucle pour l'application `somme_premiers_impairs(5)` donc pour `n=5`.

Exercice 3.2 : Fonction mystère (corrigé)

Le but de cet exercice est de réussir à déterminer ce que fait une fonction, sans en connaître ni le nom, ni la spécification mais simplement en étudiant son implémentation.

Soit la fonction «mystère» `f` ci-dessous :

```python
def f(x, y):
    """
    ??? mystère !"""

    # z : ?
    z = 0

    # w : ?
    w = x

    while w <= y:
        z = z + w * w
        w = w + 1

    return z
```

Question 1

Compléter cette définition en donnant la signature de la fonction ainsi que les types à déclarer pour les variables, en supposant que les paramètres `x` et `y` sont entiers.

Question 2

Selon les principes vus en cours, effectuer une *simulation de boucle* correspondant à l'évaluation de l'application :

```python
f(3, 6)
```

donc pour `x=3` et `y=6`.

Quelle est la valeur retournée par cette application ?

Question 3

En supposant qu'on évalue `mystere(x, y)` avec $x \leq y$. Pour quelle valeur de w, la boucle va-t-elle s'arrêter ?

Question 4

Que pensez-vous de l'application `f(5, 3)` ?

Quelle est la valeur retournée ? En déduire une *hypothèse d'appel* pertinente pour la fonction mystère.

Question 5

En déduire une définition complète et plus lisible de cette fonction, en particulier :

- proposer un nom plus pertinent pour la fonction
- renommer les paramètres et expliquer leur rôle dans la description de la fonction
- renommer les variables et expliquer leur rôle dans le corps de la fonction
- proposer un jeu de tests pour valider la fonction.

Exercice 3.3 : Nombres premiers (corrigé)

Le but de cet exercice est d'écrire une fonction qui teste si un nombre est premier ou non. Ceci permet d'étudier les alternatives *dans* les boucles ainsi que les sorties anticipées de fonction.

On rappelle qu'un entier naturel n est dit *premier* s'il n'existe aucun entier naturel autre que 1 et n lui-même qui le divise. Par convention, 1 n'est *pas* un nombre premier.

Question 1

Donner une définition de la fonction `divise` qui, étant donné un entier naturel non nul n et un entier naturel p renvoie `True` si n divise p, `False` sinon.

Par exemple :

```
>>> divise(1, 4)
True
```

```
>>> divise(2, 4)
True
```

```
>>> divise(3,4)
False
```

```
>>> divise(4,2)
False
```

Question 2

On se propose de définir la fonction `est_premier` qui, étant donné un entier naturel n, renvoie `True` si n est premier, `False` sinon.

Par exemple :

```
>>> est_premier(0)
False
```

```
>>> est_premier(1)
False
```

```
>>> est_premier(2)
True
```

```
>>> est_premier(17)
True
```

```
>>> est_premier(357)
False
```

Donner *deux* définitions *distinctes* de la fonction `est_premier` :

- une définition *sans* sortie anticipée de la fonction
- une définition *avec* sortie anticipée

Exercice 3.4 : Calcul du PPCM

Cet exercice sur le plus petit commun multiple illustre la phase de réflexion préalable nécessaire à la résolution d'un problème de calcul répétitif.

Question 1

Sans utiliser l'opérateur `%` ni l'opérateur `//`, donner une définition de la fonction `reste` qui, étant donné un entier naturel `a` et un entier naturel `b` non nul, renvoie le reste de la division euclidienne de `a` par `b`.

Par exemple :

```
>>> reste(11, 4)
3
```

```
>>> reste(21, 7)
0
```

```
>>> reste(0, 3)
0
```

Question 2

Donner une définition de la fonction `est_divisible` qui, étant donné un entier naturel `a` et un entier naturel `b` non nul, renvoie la valeur `True` si `a` est divisible par `b` et la valeur `False` sinon.

Par exemple :

```
>>> est_divisible(11, 4)
False
```

```
>>> est_divisible(21, 7)
True
```

```
>>> est_divisible(0, 3)
True
```

Question 3

Donner une définition de la fonction `ppcm` qui, étant donné deux entiers naturels `a` et `b` non nuls, renvoie le plus petit entier naturel non nul qui est un multiple commun à `a` et `b`.

Par exemple :

```
>>> ppcm(2, 3)
6
```

```
>>> ppcm(6, 8)
24
```

```
>>> ppcm(12, 15)
60
```

Question 4

Effectuer la simulation de boucle pour l'application `ppcm(6, 8)` donc pour `a=6` et `b=8`.

Exercice 3.5 : Suite de Fibonacci

Dans cet exercice, nous nous intéressons aux calculs répétitifs autour des nombres de la *suite de Fibonacci*.

Les termes de la suite de *Fibonacci* sont définis par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{pour tout } n > 1 \end{cases}$$

Question 1

Proposer une définition de la fonction `fibonacci` qui calcul le `n`-ième terme de la suite de *Fibonacci*.

Les premiers termes de cette suite sont les suivants :

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
0	1	1	2	3	5	8	13	21

Remarque : il sera utile de séparer les cas `n == 0` et `n >= 1`. On pourra également avoir besoin d'une variable temporaire `temp` pour stocker un résultat intermédiaire.

Question 2

Effectuer la simulation de la boucle du cas `n >= 2` pour `fibonacci(8)`.

Quelle est valeur de F_8 ?

Question 3

La suite de *Fibonacci* possède de nombreuses propriétés intéressantes. L'une de ces propriétés lui donne même un côté un peu *mystique*.

Commençons donc à étudier cette propriété qui repose sur les divisions entre éléments consécutifs de la suite de *Fibonacci*, donc la suite :

$$D_k = \frac{F_k}{F_{k-1}} \text{ pour } k \geq 2$$

Donner une définition de la fonction `fibo_diff` qui retourne le `k`-ième terme de la suite D.

Donner un jeu de tests pour les trois premiers termes de la suite.

Question 4

Voici quelques valeurs de `fibo_diff` pour $k \geq 5$:

```
>>> fibo_diff(5)
1.6666666666666667
```

```
>>> fibo_diff(10)
1.6176470588235294
```

```
>>> fibo_diff(41)
1.618033988749895
```

On peut montrer en fait que la suite D_k converge et tend, lorsque k tend vers l'infini, vers la valeur suivante :

$$\varphi = \frac{1 + \sqrt{5}}{2}$$

```
>>> import math
>>> (1 + math.sqrt(5)) / 2
1.618033988749895
```

Au 41-ième terme de la suite nous avons déjà atteint la précision maximale pour l'approximation de la constante φ qui n'est autre que le célèbre **nombre d'or**, cher aux architectes, peintres, etc.

Cette propriété de la suite D_k de converger vers le nombre d'or φ nous offre un moyen alternatif pour calculer le `n`-ième terme de la suite de *Fibonacci*.

L'idée est d'exploiter la formule :

$$F_n \approx \frac{\varphi^n}{\sqrt{5}}$$

lorsque n est «suffisamment» grand.

En utilisant l'opérateur `**` d'élévation à la puissance de Python, proposer une définition de la fonction `fibo_approx` qui utilise la formule précédente pour approcher les éléments de la suite de *Fibonacci*.

Remarque : il n'est pas évident de tester cette fonction, mais on peut donner les exemples suivants :

```
>>> fibo_approx(5)
4.959674775249769
```

```
>>> fibonacci(5)
5
```

```
>>> fibo_approx(10)
55.00363612324743
```

```
>>> fibonacci(10)
55
```

On peut voir sur ces exemples que l'approximation est plutôt bonne même pour des valeurs assez «petites» de `n`.

Que proposez-vous pour tester votre fonction d'approximation ?

Indice : l'arrondi d'un flottant `x` à l'entier le plus proche (vers `0`) se note `round(x)` en Python.

Complément : la page Wikipedia sur le thème *Suite de Fibonacci* contient de nombreuses informations intéressantes sur ce thème.

Exercice 3.6 : Encadrements

Le but de cet exercice est de mettre en œuvre une boucle `while` utilisant une expression booléenne complexe comme condition.

Question 1 : la partie entière

La partie entière d'un nombre réel positif x est définie comme étant l'entier naturel n tel que $n \leq x < n + 1$. Cet entier est unique et, avec une calculatrice ou un programme informatique, on l'obtient généralement en utilisant une fonction mathématique prédéfinie.

Dans cet exercice, nous allons écrire une fonction capable de retourner la partie entière d'un réel positif, en utilisant une boucle pour trouver la valeur de n correspondant à x.

Donner une définition de la fonction `partie_entière` qui, étant donné un nombre réel positif x, renvoie la valeur de la partie entière de ce nombre.

Par exemple :

```
>>> partie_entiere(0.66)
0
```

```
>>> partie_entiere(2)
2
```

```
>>> partie_entiere(2.75)
2
```

Question 2 : encadrement large

La partie entière d'un nombre définit donc un encadrement d'écart 1 de ce nombre. On souhaite maintenant généraliser cela en donnant un écart d'encadrement variable.

Donner une définition de la fonction `encadrement` qui, étant donné un nombre réel positif x et un entier strictement positif *ecart*, renvoie le plus petit entier naturel b tel que $b \leq x < b + ecart$.

Par exemple :

```
>>> encadrement(0.66, 2)
0
```

```
>>> encadrement(7.42, 1)
7
```

```
>>> encadrement(7.42, 3)
5
```

```
>>> encadrement(7.42, 4)
4
```

Question 3

Donner une définition de la fonction `partie_entière2` de même spécification que la fonction `partie_entière` et qui utilise la fonction `encadrement`.

Exercice 3.7 : Approximation de la racine carrée d'un nombre

Le but de cet exercice est d'écrire une fonction qui calcule une valeur approchée de la racine carrée d'un nombre. Soit x un nombre réel positif, une valeur approchée de $\sqrt{x}$ est donnée par le calcul des valeurs de la suite suivante :

$$u_n = \begin{cases} 1 & \text{si } n = 0 \\ \frac{u_{n-1} + \frac{x}{u_{n-1}}}{2} & \text{sinon} \end{cases}$$

Question 1

Donner une définition de la fonction `suite_racine` qui, étant donné un nombre positif x et un entier naturel n, calcule la $n^{\text{ième}}$ approximation de $\sqrt{x}$ (*i.e.* la valeur de u_n comme défini ci-dessus).

Par exemple :

```
>>> suite_approx(4, 0)
1.0
```

```
>>> suite_approx(4, 1)
2.5
```

```
>>> suite_approx(4, 2)
2.05
```

```
>>> suite_approx(4, 6)
2.0
```

Bien sûr, on se rappelle que $\sqrt{4} = 2$ donc la suite d'approximation converge assez vite : elle atteint la précision flottante de Python dès le 6ème terme de la suite.

Critère d'arrêt

La fonction précédente nécessite de donner explicitement un rang auquel s'arrêter pour calculer la valeur approchée de $\sqrt{x}$. On aimerait se passer de ce critère contraignant et arrêter le calcul quand la valeur est *suffisamment* proche de la valeur exacte. Pour cela, on peut considérer deux critère d'arrêt possibles :

— on s'arrête dès que la valeur approchée calculée par la suite ne change plus (*i.e* lorsque deux valeurs consécutives u_n et u_{n-1} sont indistinguables par la machine).

— on s'arrête quand la différence entre la valeur calculée et la valeur réelle sont identiques à ϵ près, paramètre de la fonction.

Les deux prochaines questions de l'exercice consistent à mettre en œuvre ces deux solutions.

Question 2

Donner une définition de la fonction `approx_racine_stable` qui implémente la première solution, c'est-à-dire qui calcule itérativement les valeurs de la suite u_n et s'arrête dès que la valeur approchée ne change plus.

Par exemple :

```
>>> approx_racine_stable(4)
2.0
```

```
>>> approx_racine_stable(25)
5.0
```

```
>>> approx_racine_stable(2)
1.414213562373095
```

Pour ce dernier exemple, on peut vérifier que Python est «presque» d'accord avec notre approximation :

```
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
```

Question 3

Donner une définition de la fonction `approx_racine_eps` qui implémente la seconde solution, c'est à dire qui, étant donnés deux nombres positifs x et e, renvoie la valeur approchée de $\sqrt{x}$ obtenue en calculant les termes de la suite jusqu'à ce que deux valeurs consécutives ont une différence inférieure ou égale à e.

Par exemple :

```
>>> approx_racine_eps(4, 0.1)
2.000609756097561
```

```
>>> approx_racine_eps(4, 0.0001)
2.000000000000002
```

```
>>> approx_racine_eps(4, 0.00000001)
2.0
```

```
>>> approx_racine_eps(25, 0.00001)
5.0
```

```
>>> approx_racine_eps(2, 0.000001)
1.414213562373095
```

Exercice 3.8 : Lancers de dés (sur machine)

Le but de cet exercice est de mettre en œuvre des boucles `while` simples mais exploitant la *génération de nombres aléatoires*.

Question 1 : lancer un dé à 6 faces

Souvent, en informatique, il est nécessaire de simuler le hasard : dans les jeux vidéos (pour leur éviter d'être trop prévisibles) ou en sécurité informatique (cryptographie). Le «véritable» hasard étant hors de portée (informatique), on utilise des générateurs de séquences pseudo-aléatoires.

En Python, le générateur pseudo-aléatoire est fourni par le module de bibliothèque `random` et la fonction `random.random`.

```
>>> import random
```

La fonction `random.random` possède (dans sa version la plus simple) la signature :

```
 -> float
```

Donc elle ne prend pas de paramètre et retourne un flottant.

Ce flottant est le prochain nombre flottant dans la séquence pseudo-aléatoire du générateur, dans l'intervalle $[0.0; 1.0[$.

Par exemple :

```
>>> random.random()
0.7098148541342282
```

A chaque appel, cette fonction retourne une valeur aléatoire supérieure ou égale à 0.0 et strictement inférieure à 1.0.

Ainsi, si vous l'utilisez plusieurs fois sur ordinateur, vous pourrez remarquer qu'elle ne fournit «jamais» la même valeur en retour (dans le cas où ça arriverait, dites-vous que c'est encore beaucoup plus fort que de gagner à la loterie mais sans aucun gain autre que la satisfaction personnelle).

```
>>> random.random()
0.5015904299398073
```

```
>>> random.random()
0.4674017568108694
```

En utilisant la fonction `random.random`, donner une définition de la fonction `lancer_de6`, sans argument, qui simule le lancer d'un dé à 6 faces. C'est-à-dire que cette fonction renvoie à chaque appel une valeur **entière** comprise entre 1 et 6.

Indice : la fonction `math.floor(x)` rend la valeur de l'entier le plus proche de `x` qui lui est inférieur. Pour l'utiliser, il est nécessaire de réaliser l'importation du module de bibliothèque mathématique:

```
>>> import math
```

Exemples d'utilisation de la fonction `lancer_de6` (il est évident que le résultat n'est pas prévisible) :

```
>>> lancer_de6()
6
```

```
>>> lancer_de6()
5
```

```
>>> lancer_de6()
6
```

```
>>> lancer_de6()
2
```

```
>>> lancer_de6()
5
```

```
>>> lancer_de6()
2
```

```
>>> lancer_de6()
2
```

```
# Jeu de test
assert 1 <= lancer_de6() <= 6   # unique test possible
```

Question 2 : aléa et jeux de tests

A priori, on ne peut pas produire de jeu de tests pour `lancer_de6` car on ne peut pas savoir à l'avance quelles seront les valeurs retournées, sinon tout cela ne serait pas très aléatoire.

Cependant on veut parfois pouvoir reproduire une série de lancers. Pour cela on utiliser ce que l'on appelle une *graine* pour initialiser le générateur. Chaque *graine* produit une séquence prévisible qui lui est spécifique. La graine peut être précisée par la fonction `random.seed` qui accepte un entier comme graine en paramètre.

Par exemple :

```
>>> random.seed(42)
```

```
>>> random.random()
0.6394267984578837
```

```
>>> random.random()
0.025010755222666936
```

```
>>> random.random()
0.27502931836911926
```

```
>>> random.seed(42)
```

```
>>> random.random()
0.6394267984578837
```

```
>>> random.random()
0.025010755222666936
```

```
>>> random.random()
0.27502931836911926
```

Proposer un jeu de test pour `lancer_de6`.

Question 3 : moyenne d'une série de lancers

Définir la fonction `moyenne_plusieurs_lancers` qui, étant donné un entier naturel non nul n, retourne la moyenne obtenue en lançant n fois un dé à 6 faces à l'aide de la fonction précédente.

Question 4 : fréquence d'une valeur aléatoire

La fonction précédente permet de se rendre compte que la moyenne obtenue sur un grand nombre de génération de valeurs aléatoires par la fonction `lancer_de6` est identique à celle qui serait obtenue avec un dé à 6 faces classique. Par contre, ce n'est pas encore suffisant pour être certain que l'on a là une excellente simulation d'un lancer de dé. Pour cela, il va falloir vérifier que, comme pour un dé classique, la fréquence d'apparition de chaque valeur est égale à $\frac{1}{6}$.

Définir la fonction `frequence_valeur` qui, étant donné un entier naturel r, compris entre 1 et 6, et un entier naturel non nul n, retourne la fréquence d'apparition de la valeur r lors de n lancers d'un dé à 6 faces à l'aide de la fonction `lancer_de6`.

Voici quelques résultats expérimentaux :

```
>>> frequence_valeur(1, 1)
0.0
```

```
>>> frequence_valeur(1, 10)
0.2
```

```
>>> frequence_valeur(1, 100)
0.13
```

```
>>> frequence_valeur(1, 10000)
0.1685
```

```
>>> frequence_valeur(2, 10000)
0.1655
```

```
>>> frequence_valeur(3, 10000)
0.1739
```

```
>>> frequence_valeur(4, 10000)
0.1695
```

```
>>> frequence_valeur(5, 10000)
0.1618
```

```
>>> frequence_valeur(6, 10000)
0.1631
```

```
>>> 1/6
0.16666666666666666
```

Question subsidiaire: que pouvez-vous en conclure ? Est-ce que la génération des valeurs aléatoires par la fonction `lancer_de6` suit bien une loi uniforme ?

Question 5 : lancer de dé à n faces

Donner une définition de la fonction `lancer_deN` qui, étant donné un entier naturel non nul n simule le lancer d'un dé à n faces. C'est-à-dire que cette fonction renvoie à chaque appel une valeur **entière** comprise entre 1 et n.

Par exemple :

```
>>> lancer_deN(6)
2
```

```
>>> lancer_deN(10)
6
```

```
>>> lancer_deN(20)
2
```

```
>>> lancer_deN(20)
5
```

```
>>> lancer_deN(20)
14
```

```
>>> lancer_deN(100)
55
```

```
>>> lancer_deN(100)
23
```

```
# Jeu de tests
assert 1 <= lancer_deN(20) <= 20
assert 1 <= lancer_deN(30) <= 30
assert lancer_deN(1) == 1
```

Thème 4 : Exercices avancés sur les boucles

Exercice 4.1 : Retour sur la factorielle (corrigé)

Cet exercice reprend la définition de la factorielle vue précédement et étudie les problèmes de correction et de terminaison la concernant. On rappelle ci-dessous la définition proposée :

```python
def factorielle(n : int) -> int:
    """Précondition : n >= 0
    Retourne le produit factoriel n!
    """
    # Rang
    k : int = 1

    # Factorielle au rang k
    f : int = 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

Question 1

Proposer un jeu de tests pour valider la fonction `factorielle`.

Question 2

Effectuer la simulation de boucle pour l'application `factorielle(5)`.

Question 3 : à propos de la correction

Afin de vérifier que la définition proposée ci-dessus réalise correctement le calcul de la factorielle :

— proposer un **invariant de boucle**
— vérifier cet invariant sur la simulation de boucle pour `factorielle(5)`.
— en supposant que cet invariant est vérifié pour n'importe quelle valeur de `n` satisfaisant l'hypothèse `n > 0`, justifier le fait que `factorielle(n)` calcule bien $n!$.

Question 4 : à propos de la terminaison

Afin d'assurer la terminaison de la fonction `factorielle` :

— proposer un **variant de boucle**
— vérifier ce variant sur la simulation de boucle `factorielle(5)`
— justifier informellement le fait que `factorielle(n)` termine toujours.

Exercice 4.2 : Fonction mystère (corrigé)

Le but de cet exercice est de réussir à déterminer ce que calcule une fonction mystère en simulant des boucles imbriquées.

Considérer la fonction «mystère» `f` suivante:

```
def f(n,m):
    a=n
    b=0
    c=0

    while a>0:
        while a>0:
            a = a-1
            b = b+1
        a = b-1
        b = 0
        c = c+m
    return c
```

Question 1

Compléter la définition ci-dessus avec la signature de la fonction ainsi que les types dans les déclarations de variables.

Question 2

Calculer "à la main" différentes valeurs de `f` (sur de petits entiers).

On explique ici comment effectuer une simulation de boucles imbriquées, avec l'appel de `f(3,4)`.

Les variables `a`, `b` et `c` sont modifiées dans cet ordre à chaque tour de boucle, ce seront donc les colonnes principales de notre simulation.

Dans le cas de deux boucles imbriquées, on distingue la boucle *extérieure* et la boucle *intérieure*.

On construit un tableau où la première colonne indique à quel tour de la boucle extérieure l'on se trouve et la deuxième colonne indique à quel tour de la boucle intérieure l'on se trouve (ou «-») quand on est en dehors de cette boucle). Les valeurs des variables sont celles en fin de tour de boucle comme pour les simulations «simples».

tour de boucle ext.	tour de boucle int.	a	b	c
entrée	-	**3**	**0**	**0**
	entrée	**3**	**0**	**0**
	1er	2	1	0
	2e	1	2	0
	3e (sortie)	0	3	0
1er	-	2	0	4
	entrée	**2**	**0**	**4**
	1er	1	1	4
	2e (sortie)	0	2	4
2e	-	1	0	8
	entrée	**1**	**0**	8

tour de boucle ext.	tour de boucle int.	a	b	c
	1er (sortie)	0	1	8
3e (sortie)	-	**0**	**0**	**12**

Expliquer comment interpréter cette simulation.

Question 3

Conjecturer ce que calcule `f`.

Que pensez-vous de :

- `f(3, -4)` ?
- `f(-3, 4)` ?

En déduire une définition complète de la fonction, en lui trouvant un nom plus explicite.

Exercice 4.3 : Retour sur l'algorithme d'Euclide (corrigé)

Cet exercice reprend la définition de l'algorithme d'Euclide vu au chapitre 3 et étudie les problèmes de correction et de terminaison la concernant.

La définition proposée en cours est rappelée ci-dessous :

```python
def pgcd(a : int, b : int) -> int:
    """Précondition: b > 0 et a >= b
    Retourne le plus grand commun diviseur de a et b."""

    # Quotient
    q : int = a

    # Diviseur
    r : int = b

    # Variable temporaire
    temp : int = 0

    while r != 0:
        temp = q % r
        q = r
        r = temp

    return q
```

Question 1 - Simulation

Effectuer une simuation de `pgcd(54,15)` donc pour `a=54` et `b=15`.

Question 2 - Correction

On discute maintenant de la correction de la fonction `pgcd`. Le *candidat* invariant de boucle que nous proposons est le suivant :

Candidat invariant de boucle : $\mathbf{div}(a, b) = \mathbf{div}(q, r) \wedge (q \geq r \geq 0)$, où $\mathbf{div}(x, y)$ est l'ensemble des diviseurs communs à x et y.

— vérifier cet invariant sur la simulation de `pgcd(54,15)`.
— en supposant que cet invariant est vérifié pour n'importe quelle valeur de `a, b` satisfaisant l'hypothèse $a \geq b > 0$, justifier le fait que `pgcd(a, b)` calcule bien le pgcd de `a` et `b`.

Question 3 - Terminaison

Discuter de la terminaison de la fonction `pgcd` :

— proposer un **variant de boucle**
— vérifier ce variant sur la simulation de boucle `pgcd(54,15)`
— justifier informellement le fait que `pgcd(a,b)` termine toujours.

Exercice 4.4 : Somme des cubes

Cet exercice illustre le fait qu'une même boucle peut être vue de plusieurs façons différentes : chaque vue correspondant à un invariant de boucle spécifique. On discute également d'efficacité.

Question 1

Compléter la définition de la fonction `somme_cubes` ci-dessous, qui calcule la somme :

$$\sum_{k=1}^{n} k^3$$

Remarque : pour votre jeu de tests on vérifiera les cas pour `n` dans l'intervalle $[0; 4]$.

```python
def somme_cubes(n : ??) -> ??:
    """
        ??

        retourne ???
    """
    # ?? à quoi sert s ?
    s : ?? = 0

    ??

    while k <= n:
        s = ??
        ??

    return s
```

Question 2

Proposer un invariant de boucle qui semble le plus naturel et le vérifier sur la simulation `somme_cubes(4)`.

Justifier la correction de la boucle si on suppose cet invariant vrai pour n'importe quelle valeur de n.

Question 3

Combien d'opérations arithmétiques (additions et multiplications) faut-il effectuer pour calculer `somme_cubes(n)` ?

Question 4

Il est important de noter que pour une même boucle plusieurs invariants intéressants sont possibles.

On propose donc (a priori) un autre candidat :

invariant de boucle : $s = \left(\frac{k\times(k-1)}{2}\right)^2$

Remarque : cet invariant n'est pas choisi au hasard, il utilise les propriétés mathématiques suivantes:

- pour tout entier naturel $k \geq 1$: $0^3 + 1^3 + 2^3 + ... + (k-1)^3 = (0 + 1 + 2 + ... + (k-1))^2$
- pour tout entier naturel $k \geq 1$: $0 + 1 + 2 + ... + (k-1) = \frac{k\times(k-1)}{2}$

(ces propriétés sont connues, mais elles peuvent aussi se retrouver assez aisément).

Vérifier cet invariant, toujours sur la simulation `somme_cubes(4)`.

Quelle est l'expression de l'invariant en sortie de boucle ?

Question 5

En déduire une définition de la fonction `somme_cubes_rapide` qui effectue le même calcul que `somme_cubes` mais de manière plus efficace.

Combien d'opérations arithmétiques sont nécessaires pour calculer `somme_cubes_rapide(n)` ?

Remarque : dans le jeu de test on exploitera le lien entre `somme_cubes_rapide` et `somme_cubes`.

Exercice 4.5 : Couples et intervalles

Cet exercice se concentre sur l'utilisation des boucles imbriquées ainsi que sur la notion de sorties de boucle et de fonction anticipées. On s'intéresse pour ce faire aux couples d'entiers appartenant à un intervalle.

Question 1

Donner une définition de la fonction `nb_couples_intervalle` qui, étant donné deux entiers n et p tels que $n \leq p$, renvoie le nombre de couples (i, j) d'entiers appartenant à l'intervalle $[n, p]$ tels que $i < j$.

Par exemple :

```
>>> nb_couples_intervalle(0, 0)
0
```

```
>>> nb_couples_intervalle(2, 4)
3
```

```
>>> nb_couples_intervalle(-1, 3)
10
```

Question 2

Donner une définition de la fonction `nb_couple_divise` qui, étant donné deux entiers n et p tels que $n \leq p$, compte le nombre de couples (i, j) d'entiers distincts appartenant à l'intervalle $[n, p]$ tels que i divise j.

Par exemple :

```
>>> nb_couples_divise(4,6)
0
```

```
>>> nb_couples_divise(2,6)
3
```

```
>>> nb_couples_divise(-1,1)
2
```

```
>>> nb_couples_divise(1,10)
17
```

Question 3 : tracer une exécution (TME)

Modifier la fonction précédente pour qu'elle trace l'exécution des boucles imbriquées. Il faut pour cela insérer des instructions d'affichage (`print`) au bon endroit.

Par exemple :

```
>>> nb_couples_divise_trace(2,6)
couple ( 2 , 3 )
couple ( 2 , 4 )
------------
2 divise 4 !
------------
couple ( 2 , 5 )
couple ( 2 , 6 )
------------
2 divise 6 !
------------
couple ( 3 , 4 )
couple ( 3 , 5 )
couple ( 3 , 6 )
------------
3 divise 6 !
------------
couple ( 4 , 5 )
couple ( 4 , 6 )
couple ( 5 , 6 )
3
```

Question 4 : Sortie de boucle anticipée

On se pose maintenant le problème non pas du nombre mais de l'existence d'un tel couple (i, j) tel que i divise j.

Une solution possible pour répondre à ce problème pourrait être la définition de la fonction suivante :

```
def existe_couples_divise(n : int, p : int) -> bool:
    """Précondition : n <= p
    Renvoie True s'il existe un couple (i,j) d'entiers appartenant
            à l'intervalle [n,p] tels que i != j et i divise j,
            ou False sinon.
    """
    return nb_couples_divise(n,p) > 0
```

```
# Jeu de tests
assert existe_couples_divise(0, 0) == False
assert existe_couples_divise(2, 6) == True
assert existe_couples_divise(-1, 1) == True
assert existe_couples_divise(1, 10) == True
assert existe_couples_divise(21, 34) == False
```

Le problème de cette définition est qu'elle n'est pas efficace puisqu'elle calcule tous les couples possibles alors qu'il suffit de s'arrêter dès qu'un couple est trouvé.

Donner une définition de la fonction `existe_couples_divise_rapide` qui utilise une *sortie de boucle anticipée* pour améliorer l'efficacité de la fonction.

Question 5 : Sortie de fonction anticipée

Donner une variante de cette fonction utilisant une *sortie de fonction anticipée*.

Exercice 4.6 : Combinaisons

Cet exercice est un exercice d'ouverture. Il s'intéresse à la notion d'efficacité sur le problème classique du comptage de combinaisons. Il illustre en particulier qu'un même problème peut être résolu de différentes façons, certaines plus efficaces que d'autres.

La fonction factorielle est un grand classique des cours d'introduction à la programmation. Nous la retrouvons d'ailleurs en cours et dans d'autres exercices. Cependant la factorielle n'est pas juste un outil pédagogique, elle joue un rôle important pour le comptage en combinatoire.

Soit un ensemble E de taille n, le nombre de permutations d'éléments de E est exactement $n!$.

Il y a par exemple $5! = 120$ façons de classer 5 livres différents.

La définition proposée en cours pour la fonction `factorielle` (sans son jeu de tests) est la suivante :

```
def factorielle(n : int) -> int:
    """Précondition : n >= 0
    Retourne le produit factoriel n!
    """
    # on démarre au rang 1
```

```python
    k : int = 1

    # factorielle au rang k
    f : int = 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

Par exemple :

```python
>>> factorielle(5)
120
```

Question 1

On souhaite compter le nombre d'opérations arithmétiques nécessaires pour calculer la factorielle. Pour cela, définir une variante `factorielle_compte` de la fonction précédente en effectuant les modifications suivantes :

— ajouter une variable `nb_ops` permettant de compter les multiplications effectuées par une applicationde `factorielle(n)` (on ne s'intéresse pas aux incrémentations du compteur de boucle `k`).
— afficher avant le retour de la variable ce nombre d'opération (avec l'instruction `print`).

Par exemple :

```python
>>> factorielle_compte(5)
Nombre d'opérations =  5
120
```

Combien de multiplications effectue le calcul de `factorielle(n)` ?

Question 2

Une *combinaison* de k parmi n (avec $k \leq n$) correspond au nombre de façons de choisir k éléments dans un ensemble E de taille n.

Ce nombre est noté $\binom{n}{k}$ au niveau international (mais souvent C_n^k en France, $C_{n,k}$ en Italie et C_k^n dans d'autres pays !). Ce nombre est également appelé *coefficient binomial*.

Par exemple, il y a $\binom{5}{3} = 10$ façon de choisir 3 livres parmi 5.

Il est facile de trouver une première formulation pour le nombre $\binom{n}{k}$:

— on prend tout d'abord le maximum de n permutations possibles, soit $n!$
— on élimine toutes les permutations des k éléments choisis (le choix ne permute pas). Il y a $k!$ permutations à éliminer et on obtient donc $\frac{n!}{k!}$.
— mais une fois le choix effectué, les $n - k$ éléments restants ne doivent pas permuter indépendamment des k éléments choisis, on doit donc éliminer $(n - k)!$ permutations.

On arrive donc à la formule bien connue : $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

En se basant sur cette formule, proposer une définition de la fonction `combinaisons` qui, étant donné deux entiers `n` et `k`, retourne le nombre $\binom{n}{k}$.

Question 3

Quel est le nombre de multiplications effectuées pour calculer :

- `combinaisons(5, 2)`
- `combinaisons(10, 4)`
- `combinaisons(1000, 2)`
- `combinaisons(10000, 450)`

Question 4

On se propose de résoudre le problème des combinaisons de façon plus efficace en remarquant que de nombreuses multiplications sont redondantes.

Par exemple : $\binom{5}{3} = \frac{5!}{3!2!} = \frac{5\times4\times3\times2\times1}{3\times2\times1\times2\times1} = \frac{5\times4}{2} = 10$.

Dans la dernière étape de ce calcul nous n'avons eu besoin que d'une seule multiplication, à comparer aux 10 de la formule initiale. On doit pouvoir factoriser certains calculs en exploitant la formule ci-dessous :

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-(k-1))}{k(k-1)(k-2)\cdots 1} = \prod_{i=1}^{k} \frac{n-(k-i)}{i} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

Donner une définition de la fonction `combis_rapide` telle que `combis_rapide(n, k)` retourne $\binom{n}{k}$ selon la formule ci-dessus.

Question 5

En effectuant des simulations, compter le nombre de multiplications et de divisions réalisées par la fonction `combis_rapide` sur les deux applications suivantes :

- `combis_rapide(10, 4)`
- `combis_rapide(1000, 2)`

Thème 5 : Exercices sur les intervalles et chaînes de caractères

---

Exercice 5.1 : Intervalles (corrigé)

Cet exercice permet de se familiariser avec les intervalles (type `range`) et les boucles d'itérations avec `for ... in ...`.

Les réponses aux questions doivent donc impérativement exploiter ces constructions.

Question 1

Donner une définition de la fonction `somme_carres` qui, étant donné un entier naturel `n`, retourne la somme des carrés des nombres entiers inférieurs ou égaux à `n`.

Question 2

Soit la fonction mystere `f` suivante :

```python
def f(m, n):
    """
        ???
    """

    p : int = 1

    k : int = m
    while k < n:
        p = p * k * k * k
        k = k+1

    return p
```

Expliquer ce que fait cette fonction et, en complément, donner une définition mathématique de ce calcul et compléter sa spécification.

En déduire une définition complète (avec spécification et jeu de tests) de cette fonction mystère (renommée pour l'occasion) en utilisant une boucle `for ... in ...` en remplacement de `while`.

Effectuer une simulation de votre fonction pour `m=4` et `n=8`.

Exercice 5.2 : Fonction mystère (corrigé)

Le but de cet exercice est de réussir à déterminer ce que calcule une fonction mystère en simulant une boucle for.

Considérer la fonction «mystère» `f` suivante:

```python
def f(a):
    b = 0
    for c in a:
```

```
        if c >= '0' and c <= '9':
            b = b + 1

    return b
```

Question 1

Compléter la définition ci-dessus avec la signature de la fonction ainsi que les déclarations de variables.

Question 2

Effectuer une simulation de boucle correspondant à l'évaluation de l'application `f('10 août')`

Calculer «à la main» les valeurs de `f` pour 'bonjour', 'un : 1', '606060'.

Question 3

Conjecturer ce que calcule `f`.

En déduire une définition complétée de la fonction, en lui trouvant un nom plus explicite.

Exercice 5.3 : Palindromes (corrigé)

Question 1

Donnez la spécification et une définition de la fonction `est_palindrome` telle que `est_palindrome(s)` retourne `True` si `s` est un palindrome, c'est-à-dire une chaîne qui est la même si on la lit de la gauche vers la droite ou de la droite vers la gauche.

Par exemple :

```
>>> est_palindrome('')
True
```

```
>>> est_palindrome('je ne suis pas un palindrome')
False
```

```
>>> est_palindrome('aba')
True
```

```
>>> est_palindrome('amanaplanacanalpanama')
True
```

Question 2

On se propose maintenant de définir une fonction de création automatique de palindromes.

Cette fonction `miroir` prend une chaîne de caractères en paramètre et retourne un palindrome à partir de cette chaîne, correspondant simplement au miroir de la chaîne.

Par exemple :

```
>>> miroir('abc')
'abccba'
```

```
>>> miroir('amanaplanacanal')
'amanaplanacanallanacanalpanama'
```

```
>>> miroir('do-re-mi-fa-sol')
'do-re-mi-fa-sollos-af-im-er-od'
```

Exercice 5.4 : Suppressions (corrigé)

Cet exercice propose des variantes de la fonction `suppression` donnée comme exemple de filtrage dans le présent chapitre.

Question 1

Donner une définition de la fonction `suppression_debut` telle que `suppression_debut(c,s)` supprime la *première* occurrence du caractère `c` dans la chaîne `s`.

Par exemple :

```
>>> suppression_debut('a', '')
''
```

```
>>> suppression_debut('a', 'aaaaa')
'aaaa'
```

```
>>> suppression_debut('p', 'le papa noel')
'le apa noel'
```

```
>>> suppression_debut('a', 'bbbbb')
'bbbbb'
```

Question 2

Donner la spécification et une définition de la fonction `suppression_derniere` telle que `suppression_derniere(c, s)` supprime la *dernière* occurrence du caractère `c` dans la chaîne `s`.

Par exemple :

```
>>> suppression_derniere('a','')
''
```

```
>>> suppression_derniere('a', 'aaaaa')
'aaaa'
```

```
>>> suppression_derniere('p', 'le papa noel')
'le paa noel'
```

```
>>> suppression_derniere('a', 'bbbbb')
'bbbbb'
```

Exercice 5.5 : Voyelles

Cet exercice étudie des réductions, filtrages et transformations simples de chaînes. On s'intéresse aux voyelles présentes dans une chaîne de caractère.

Pour identifier une voyelle, on utilise le prédicat suivant :

```python
def est_voyelle(c : str) -> bool:
    """Précondition : len(c) == 1
    Retourne True si et seulement si c est une voyelle
             miniscule ou majuscule.
    """
    return (c == 'a') or (c == 'A') \
        or (c == 'e') or (c == 'E') \
        or (c == 'i') or (c == 'I') \
        or (c == 'o') or (c == 'O') \
        or (c == 'u') or (c == 'U') \
        or (c == 'y') or (c == 'Y')
```

```python
# Jeu de tests
assert est_voyelle('a') == True
assert est_voyelle('E') == True
assert est_voyelle('b') == False
assert est_voyelle('y') == True
assert est_voyelle('z') == False
```

Question 1 : réduction

Donner une définition de la fonction `nb_voyelles` qui retourne le nombre de voyelles présentes dans une chaîne `s` passée en paramètre. Il s'agit donc d'une réduction d'une chaîne vers le type `int`.

Par exemple :

```python
>>> nb_voyelles('la maman du petit enfant le console')
12
```

```python
>>> nb_voyelles('mr brrxcx')
0
```

```python
>>> nb_voyelles('ai al o ents')
5
```

Question 2 : accents

Considérons l'application suivante :

```python
>>> nb_voyelles('la maman du bébé le réconforte')
8
```

On peut remarquer que les lettres accentuées ne sont pas prises en compte dans le compte des voyelles. Proposer une définition de la fonction `nb_voyelles_accents` qui corrige ce défaut.

Par exemple :

```
>>> nb_voyelles_accents('la maman du bébé le réconforte')
11
```

Question 3 : filtrage

Donner une définition de la fonction de filtrage `sans_voyelle` qui élimine les voyelles d'une chaîne de caractères.

Par exemple :

```
>>> sans_voyelle('aeiouy')
''
```

```
>>> sans_voyelle('la balle au bond rebondit')
'l bll  bnd rbndt'
```

```
>>> sans_voyelle('mr brrxcx')
'mr brrxcx'
```

Question 4 : transformation

Donner une définition de la fonction `mot_mystere` qui remplace dans une chaîne de caractères les voyelles par des symboles *soulignés* `_`.

Par exemple :

```
>>> mot_mystere('aeiouy')
'______'
```

```
>>> mot_mystere('la balle au bond rebondit bien')
'l_ b_ll_ __ b_nd r_b_nd_t b__n'
```

```
>>> mot_mystere('mr brrxcx')
'mr brrxcx'
```

Exercice 5.6 : Brins d'ADN

Cet exercice s'intéresse à la manipulation des chaînes de caractères et aborde notamment l'opération de découpage ainsi que la notion de fonctions partielles vues en cours.

Un brin d'ADN (acide désoxyribonucléique) est une molécule présente dans tout organisme vivant et caractérisée en général par une séquence de bases azotées. Il existe 4 bases différentes : l'adénine, la thymine, la cytosine, et la guanine. Ces quatre bases jouent un rôle important chez les organismes vivants puisque c'est leur agencement (c'est à dire leur ordre dans la séquence) qui détermine le rôle du brin d'ADN.

Dans cet exercice, les 4 bases seront représentées par les 4 caractères "A", "T", "C" et "G" et les brins d'ADN par des chaînes de caractères. Ainsi un brin d'ADN qui présente la séquence adénine, adénine, cytosine, guanine sera représenté par la chaîne de caractère "AACG"

Question 1

Chacune des quatre bases possède une base dite *complémentaire* avec laquelle elle est capable de s'apparier. Ainsi l'adénine et la thymine sont complémentaires et la cytosine et la guanine sont complémentaires.

Donner une définition de la fonction `base_comp` qui, étant donnée une base azotée, renvoie la base complémentaire. Par exemple :

```
>>> base_comp('A')
'T'
```

```
>>> base_comp('G')
'C'
```

```
>>> base_comp('T')
'A'
```

```
>>> base_comp('C')
'G'
```

Question 2

Par extension le brin complémentaire d'un brin d'ADN est constitué d'une séquence de bases de même longueur mais contenant les bases complémentaires de celui-ci, **dans l'ordre inverse**.

Donner une définition de la fonction `brin_comp` qui, étant donné un brin d'ADN, renvoie le brin d'ADN complémentaire.

Par exemple :

```
>>> brin_comp('ATCG')
'CGAT'
```

```
>>> brin_comp('ATTGCCGTATGTATTGCGCT')
'AGCGCAATACATACGGCAAT'
```

```
>>> brin_comp('')
''
```

Question 3

Donner une définition `test_comp` qui, étant donné deux brins d'ADN, teste si ces deux brins sont complémentaires. On peut remarquer qu'une condition minimale pour qu'ils soient complémentaires est qu'ils aient la même longueur.

Par exemple :

```
test_comp('','')
True
```

```
test_comp('', 'ATCG')
False
```

```
test_comp('ATCG', '')
False
```

```
test_comp('ATCG', 'CGAT')
True
```

```
test_comp('ATCG', 'TAAG')
False
```

```
test_comp('ATTGCCGTATGTATTGCGCT', 'AGCGCAATACATACGGCAAT')
True
```

Remarque : il est intéressant de comparer ces deux solutions. La première nécessite une variable supplémentaire, mais elle est formellement plus simple car elle ne nécessite pas l'instruction de contrôle `return`. C'est donc la solution à considérer si l'on veut en étudier la correction ou la terminaison. Un programmeur aguerri privilégiera la seconde solution car elle est plus concise.

Question 4 : Sous-séquence et découpage de chaînes

Une des problématiques importantes dans le domaine de l'analyse des molécules d'ADN est la recherche de séquences spécifiques.

Dans cette question, on cherche à écrire une fonction qui teste si un brin d'ADN est une sous-séquence du second, c'est à dire si sa séquence apparaît dans le second brin. Pour cela, on exploitera l'opération de découpage des chaînes de caractères vu en cours. La solution consiste, étant donné un premier brin `b1` de longueur n, à tester toutes les sous-chaînes de longueur n d'un second brin `b2`. Si l'une de ces sous-chaines est identique à `b1`, alors `b1` est une sous-séquence de `b2`. Par convention, la séqunce vide est sous-séquence de toute séquence.

Donner une définition de la fonction `test_sous_sequence` qui étant donnés deux brins d'ADN, teste si le premier est une sous-sequence du second. Par exemple :

```
>>> test_sous_sequence('','')
True
```

```
>>> test_sous_sequence('','ATCG')
True
```

```
>>> test_sous_sequence('ATCG','')
False
```

```
>>> test_sous_sequence('GC', 'TAGC')
True
```

```
>>> test_sous_sequence('GC', 'TAAG')
False
```

```
>>> test_sous_sequence('CA','TAACGGCATACATAACGCGA')
True
```

Question 5 : Fonctions partielles

Outre le fait de savoir si une séquence apparaît dans un brin d'ADN, on aimerait également connaître sa position dans le brin. Sur le principe, c'est une simple variation du problème précédent dans lequel on renvoie un entier (l'indice du début de la séquence dans le brin) au lieu d'un booléen. Mais une difficulté apparaît ici puisque si le premier brin n'est pas une sous-séquence du second, alors il n'y a pas d'indice à renvoyer. On exploitera donc la solution vue en cours qui s'appuie sur les *fonctions partielles*.

Donner une définition de la fonction `recherche_sous_sequence` qui étant donnés deux brins d'ADN `b1` et `b2`, renvoie l'indice de `b2` correspondant au début de `b1` si `b1` est une sous-séquence de `b2` et ne renvoie rien (`None`) sinon. Par exemple :

```
>>> recherche_sous_sequence('','')
0
```

```
>>> recherche_sous_sequence('','ATCG')
0
```

```
>>> recherche_sous_sequence('ATCG','')
```

```
>>> recherche_sous_sequence('GC', 'TAGC')
2
```

```
>>> recherche_sous_sequence('GC', 'TAAC')
```

```
>>> recherche_sous_sequence('CATA','TAACGGCATACATAACGCGA')
6
```

Exercice 5.7 : Conversions

Cet exercice étudie le problème classique de conversion d'un entier en chaîne de caractères, et *vice-versa*. Cela conduit à un mélange intéressant entre un problème sur les chaînes et un problème sur les entiers.

Question 1 : caractère vers chiffre

Un caractère est représenté par son numéro *Unicode* que l'on peut récupérer par la fonction primitive `ord`.

Par exemple :

```
>>> ord('0')
48
```

```
>>> ord('1')
49
```

```
>>> ord('9')
57
```

```
>>> ord('5') - ord('0')
5
```

On le devine avec ces exemples, l'une des propriétés de ces numéros *Unicode* est que les numéros codant les chiffres sont consécutifs.

À partir de cette remarque, proposer une spécification pour la fonction primitive `ord`.

En déduire une définition de la fonction de conversion `chiffre` qui, étant donné un caractère représentant un chiffre, retourne l'entier qui correspond.

Par exemple :

```
>>> chiffre('5')
5
```

```
>>> chiffre('8')
8
```

Question 2 : chaîne vers entier

Donner une définition de la fonction `entier` telle que `entier(s)` retourne l'entier représenté par la chaîne `s`.

Par exemple :

```
>>> entier('9')
9
```

```
>>> entier('42')
42
```

```
>>> entier('0')
0
```

```
>>> entier('0012')
12
```

Remarque : on pourra utiliser la fonction `chiffre` définie précédemment.

Question 3 : chiffre vers chaîne

On peut construire un caractère à partir de son numéro *Unicode* `n` en écrivant : `chr(n)` de sorte que `chr(ord(c)) = c` pour tout caractère `c`.

Par exemple :

```
>>> chr(49)
'1'
```

```
>>> chr(ord('1'))
'1'
```

```
>>> chr(8 + ord('0'))
'8'
```

```
>>> chr(4 + ord('0'))
'4'
```

Proposer une spécification pour la fonction primitive `chr`.

En déduire une définition de la fonction `caractere` qui, étant donné un chiffre n, retourne le caractère qui le représente.

Par exemple :

```
>>> caractere(8)
'8'
```

```
>>> caractere(4)
'4'
```

Question 4 : chaîne vers entier

Donner une définition de la fonction `chaine` telle que `chaine(n)` retourne la chaîne représentant l'entier naturel n.

Par exemple :

```
>>> chaine(9)
'9'
```

```
>>> chaine(42)
'42'
```

```
>>> chaine(entier('122'))
'122'
```

```
>>> entier(chaine(122))
122
```

Remarque : Ces deux fonctions de conversion existent en python et se nomment respectivement `int` (conversion d'une chaîne en un entier) et `str` (conversion de valeur, notamment un entier, en chaîne).

Exercice 5.8 : Compression

Dans cet exercice, nous illustrons un problème un peu plus complexe sur la thématique de compression de chaînes de caractères selon l'approche *run-length encoding* (RLE).

Le principe de cette compression est simple : si la chaîne contient (strictement) plus d'une occurrence successive du même caractère alors on remplace les n occurrences de ce caractère par le nombre n suivi du caractère.

Par exemple, le caractère `'c'` apparaît 3 fois d'affilée dans la chaîne `'abcccd'`, on remplace donc cette suite `'ccc'` par `'3c'` dans la chaîne et l'on obtient la chaîne compressée `'ab3cd'`.

Dans cet exercice, on suppose que les chaînes à compresser ne contiennent pas déjà des chiffres.

Par exemple :

```
>>> compression('abcccd')
'ab3cd'
```

```
>>> compression('abcccddeeeefgh')
'ab3c2d4efgh'
```

```
>>> compression('abcdefg')
'abcdefg'
```

On commence par définir une fonction permettant de reconnaître un caractère représentant un chiffre :

```
def est_chiffre(c : str) -> bool:
    """Précondition : len(c) == 1
    Retourne True si et seulement si c est un chiffre.
    """
    return ('0' <= c) and (c <= '9')
```

```python
# Jeu de tests
assert est_chiffre('4') == True
assert est_chiffre('9') == True
assert est_chiffre('x') == False
```

Question 1 : décompression

Il est plus simple d'étudier en premier l'algorithme de décompression.

L'idée de la fonction `decompression` est de transformer une sous-chaîne :

- de la forme n`c` où n est un entier et `c` un caractère différent d'un chiffre en `cccc...` composée de n répétitions du caractère `c`. En python, on peut écrire `c * n` pour répéter n fois le caractère `c`.
- ou sinon de recopier le caractère non-répété.

Par exemple :

```python
>>> decompression('ab3cd')
'abcccd'
```

```python
>>> decompression('ab3c2d4efgh')
'abcccddeeeefgh'
```

```python
>>> decompression('abcdefg')
'abcdefg'
```

Donner une définition de la fonction de décompression.

Indication : vous pouvez utiliser la fonction de conversion d'une chaîne de caractères en entier vue dans l'exercice précédent, ou bien utiliser la fonction prédéfinie `int`.

Question 2 : compression

Donner une définition de la fonction `compression` qui retourne la version compressée de la chaîne `s` passée en paramètre selon le principe RLE.

Remarque : dans le jeu de tests, on validera expérimentalement la propriété `decompression(compression(s)) = s`.

Indication : vous pouvez utiliser la fonction de conversion d'un entier en chaîne de caractères vue dans l'exercice précédent, ou bien utiliser la fonction prédéfinie `str`.

Exercice 5.9 : Anagrammes

Cet exercice utilise pour prétexte les anagrammes sur les mots, représentés en chaînes de caractères, pour faire appel à des notions d'itérations et de fonctions partielles.

Formellement, deux mots (au sens mathématique) de même longueur $a_1.a_2 \ldots a_n$ et $b_1.b_2 \ldots b_n$ sont *anagrammes* quand il existe une **permutation** σ de $[1;n]$ telle que $\forall i, b_i = a_{\sigma(i)}$.

Informellement, deux mots sont anagrammes quand l'un peut être obtenu depuis l'autre en permutant les lettres. Une conséquence importante est que deux mots sont anagrammes quand ils sont composés exactement des mêmes lettres, en comptant la multiplicité.

Par exemple `'parisien'` est anagramme de `'aspirine'`.

Dans la suite, on considère qu'un *mot* est une chaîne qui ne contient pas d'espace (la chaîne `" "`).

Question 1 - Préliminaires

Donner la définition d'une fonction partielle `moins_lettre(c,a)` qui renvoie:

- la chaîne obtenue à partir de la chaîne `c` en supprimant la première occurence de la lettre `a` dans `c` si `c` contient au moins une fois `a`,
- `None` si `c` ne contient pas `a`.

Remarque : il peut être intéressant de s'aider de l'exercice *Suppressions* pour répondre à cette question.

Question 2 - Mots anagrammes

Donner la définition d'une fonction `anagramme(m1, m2)` qui indique si les mots `m1` et `m2` sont anagrammes.

Par exemple :

```
>>> anagramme("alberteinstein", "alberteinstein")
True
```

```
>>> anagramme("alberteinstein", "riennestetabli")
True
```

```
>>> anagramme("alberteinstein", "toutestrelatif")
False
```

```
>>> anagramme("lesfeuxdelamour", "dramesexuelflou")
True
```

Indice : utiliser la fonction `moins_lettre`.

Thème 6 : Exercices simples sur les listes

Exercice 6.1 : Listes de répétitions (corrigé)

Dans cet exercice introductif, on s'intéresse à la construction de listes par répétition d'un élément ou d'une liste d'éléments.

Question 1

Donner une définition de la fonction `repetition` qui, étant donné un élément x et un entier naturel k, renvoie la liste contenant k occurrences de x.

Par exemple :

```
>>> repetition("thon", 4)
['thon', 'thon', 'thon', 'thon']
```

```
>>> repetition(3, 8)
[3, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> repetition(5, 0)
[]
```

```
>>> repetition([1, 2, 3], 5)
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

Question 2

Donner une définition de la fonction `repetition_bloc` qui, étant donné une liste `l` et un entier naturel `k`, renvoie la liste obtenue en concaténant `k` fois la liste `l`.

Par exemple :

```
>>> repetition_bloc(["chat", "thon", "loup"], 3)
['chat', 'thon', 'loup', 'chat', 'thon', 'loup', 'chat', 'thon', 'loup']
```

```
>>> repetition_bloc([1, 2, 3], 5)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> repetition_bloc([1, 2, 3, 4, 5], 0)
[]
```

Exercice 6.2 : Maximum d'une liste (corrigé)

Cet exercice propose quelques problèmes de réduction associés à la notion de maximum.

Question 1

Donner une définition de la fonction `max_liste` qui, étant donné une liste non vide de nombres, renvoie le plus grand élément de cette liste.

Par exemple :

```
>>> max_liste([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
9
```

Question 2

Donner une définition de la fonction `nb_occurrences` qui, étant donné une liste L et un élément x, renvoie le nombre d'occurrences de x dans L.

Par exemple :

```
>>> nb_occurrences([3, 7, 9, 5.4, 8.9, 9, 8.999, 5], 9)
2
```

```
>>> nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "chat")
3
```

```
>>> nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "ou")
0
```

Question 3

Donner une définition de la fonction `nb_max` qui, étant donné une liste non vide de nombres L, renvoie le nombre d'occurrences du maximum de L dans L.

Par exemple :

```
>>> nb_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
2
```

```
>>> nb_max([-2, -1, -5, -3, -1, -4, -1])
3
```

Exercice 6.3 : Liste de diviseurs (corrigé)

On construit dans cet exercice des listes d'entiers vérifiant certains prédicats (divisibilité, parité, imparité).

Question 1

Donner une définition de la fonction `liste_diviseurs` qui, étant donné un entier naturel non nul a, retourne la liste des entiers naturels qui sont diviseurs de a.

Par exemple :

```
>>> liste_diviseurs(18)
[1, 2, 3, 6, 9, 18]
```

Question 2

Donner une définition de la fonction `liste_diviseurs_impairs` qui, étant donné un entier naturel non nul a, retourne la liste des entiers naturels impairs qui sont diviseurs de a.

Par exemple :

```
>>> liste_diviseurs_impairs(24)
[1, 3]
```

```
>>> liste_diviseurs_impairs(8)
[1]
```

```
>>> liste_diviseurs_impairs(15)
[1, 3, 5, 15]
```

Exercice 6.4 : Fonction mystère (corrigé)

Il s'agit ici de déterminer ce que fait une fonction, sans en connaître ni le nom, ni la spécification mais simplement en étudiant son implémentation.

Question 1

Soit la fonction «mystère» `f` ci-dessous :

```
def f(l):

    if (len(l) == 0) or (len(l) == 1):
        return True
    else:
        for i in range(len(l) - 1):
            if l[i] >= l[i + 1]:
                return False
    return True
```

Compléter cette définition en donnant la signature de la fonction.

Question 2

Selon les principes vus en cours, effectuer une *simulation de boucle* correspondant à l'évaluation de l'application :

```
f([3, 5, 7, 10])
```

Dans cette simulation on montrera aussi les valeurs prises par `l[i]` et `l[i + 1]`.

Quelle est la valeur retournée par cette application ?

Mêmes questions pour :

```
f([3, 15, 7, 10])
```

Question 3

En déduire une définition complète et plus lisible de cette fonction, en particulier :

— proposer un nom plus pertinent pour la fonction
— compléter la description de la fonction
— proposer un jeu de tests pour valider la fonction.

Question 4

Écrire une autre définition de cette fonction, en utilisant une instruction `while` avec une sortie anticipée de boucle et non une sortie anticipée de fonction, donc *sans* utiliser `return` dans le corps de la boucle.

Exercice 6.5 : Découpages (corrigé)

Pour comprendre un mécanisme, il est utile de savoir le reconstruire. Nous appliquons ce principe dans cet exercice en reconstruisant les *découpages* de listes.

Question 1 : découpages simples

Donner une définition de la fonction `decoupage_simple` qui, étant donné une liste `l` et deux entiers `i` et `j`, retourne le découpage `l[i:j]` en faisant l'hypothèse que les indices `i` et `j` sont positifs.

Remarque : on ne peut bien sûr pas utiliser de découpages dans la définition puisque notre objectif consiste à les redéfinir.

Le jeu de tests pour cette fonction correspond aux exemples du cours sur les découpages :

```
# Jeu de tests

lcomptine : List[str]
lcomptine = ['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']

assert decoupage_simple(lcomptine, 1, 3) == lcomptine[1:3]
assert decoupage_simple(lcomptine, 3, 4) == lcomptine[3:4]
assert decoupage_simple(lcomptine, 3, 3) == lcomptine[3:3]
assert decoupage_simple(lcomptine, 5, 3) == lcomptine[5:3]
assert decoupage_simple(lcomptine, 0, 7) == lcomptine[0:7]
```

Question 2 : découpage avec pas

Donner une définition de la fonction `decoupage_pas` telle que `decoupage_pas(l, i, j, p)` retourne le même résultat que `l[i:j:p]` en supposant `i`, `j` positifs et `p` strictement positif.

Voici le jeu de tests associé :

```
# Jeu de tests
assert decoupage_pas(lcomptine,1, 5, 2) == lcomptine[1:5:2]
assert decoupage_pas(lcomptine,2, 6, 1) == lcomptine[2:6:1]
```

Question 3 : pas inverse

Donner une définition de la fonction `decoupage_pas_inv` telle que `decoupage_pas_inv(l, i, j, p)` retourne le même résultat que `l[i:j:p]` en supposant `i`, `j` positifs et `p` strictement négatif.

Voici le jeu de tests associé :

```
# Jeu de tests
assert decoupage_pas_inv(lcomptine, 5, 2, -2) == lcomptine[5:2:-2]
assert decoupage_pas_inv(lcomptine, 6, 0, -1) == lcomptine[6:0:-1]
assert decoupage_pas_inv(lcomptine, 6, 0, -3) == lcomptine[6:0:-3]
```

Question 4 : découpage généralisé

On souhaite maintenant redéfinir le découpage général `decoupage` tel quel `découpage(l,i,j,p)` retourne le même résultat que l'expression `l[i:j:p]`. La seule contrainte imposée est que `p` est différent de 0.

Il nous reste à traiter les indices négatifs, et pour cela nous utilisons la fonction de normalisation suivante.

```python
def normalisation(i : int, long: int) -> int:
    """Précondition: long >= 0
    Retourne la normalisation de l'indice k pour
    une liste de longueur long.
    """
    if i < 0: # indice négatif
        if -i <= long:  # dans l'intervalle [0;long]
            return long + i
        else: # en dehors de l'intervalle [0;long]
            return 0
    else: # indice positif
        if i > long: # en dehors de l'intervalle [0;long]
            return long
        else:
            return i  # déjà normalisé
```

```python
# jeu de tests
assert normalisation(0, 6) == 0  # positif dans [0;6]
assert normalisation(4, 6) == 4  # positif dans [0;6]
assert normalisation(6, 6) == 6  # positif dans [0;6]
assert normalisation(7, 6) == 6  # positif hors [0;6]
assert normalisation(-0, 6) == 0 # négatif (cas limite)
assert normalisation(-1, 6) == 5 # négatif dans [0;6]
assert normalisation(-3, 6) == 3 # négatif dans [0;6]
assert normalisation(-5, 6) == 1 # négatif dans [0;6]
assert normalisation(-6, 6) == 0 # négatif dans [0;6]
assert normalisation(-7, 6) == 0 # négatif hors [0;6]
```

En utilisant cette fonction ainsi que les fonction `decoupage_pas` et `decoupage_pas_inv`, proposer une définition de la fonction `decoupage` pour le découpage généralisé.

Remarque : on reprendra tous les exemples du cours pour valider la fonction.

Exercice 6.6 : Moyenne et variance

Cet exercice propose des problèmes assez simples de réduction et de transformation, sur une thématique statistique.

Question 1

Donner une définition de la fonction `somme` qui, étant donné une liste de nombres, renvoie la somme des éléments de cette liste, ou 0 si la liste est vide.

Par exemple :

```
>>> somme([1, 2, 3, 4, 5])
15
```

```
>>> somme([1, 2.5, 3.2, 4, 5])
15.7
```

```
>>> somme([1, 2.5, 3.5, 4, 5])
16.0
```

```
>>> somme([])
0
```

Question 2

Donner une définition de la fonction `moyenne` qui, étant donné une liste non vide de nombres, renvoie la moyenne des éléments de cette liste.

Par exemple :

```
>>> moyenne([1, 2, 3, 4, 5])
3.0
```

```
>>> moyenne([1, 2.5, 3.5, 4, 5])
3.2
```

```
>>> moyenne([5])
5.0
```

Question 3

Donner une définition de la fonction `carres` qui, étant donné une liste L de nombres, renvoie la liste des carrés des éléments de L.

Par exemple :

```
>>> carres([1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]
```

```
>>> carres([1, -2, -3, 4, 5])
[1, 4, 9, 16, 25]
```

```
>>> carres([])
[]
```

```
>>> carres([10, 0.5, 2.0])
[100, 0.25, 4.0]
```

```
# Jeu de tests
assert carres([1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert carres([-5, -1, 2]) == [25, 1, 4]
assert carres([]) == []
assert carres([10, 0.5]) == [100, 0.25]
```

Question 4

La *variance* d'une liste de nombres est égale à la différence entre la moyenne des carrés des éléments de la liste et le carré de la moyenne des éléments de la liste.

Donner une définition de la fonction `variance` qui, étant donné une liste non vide de nombres, renvoie la variance de la liste.

Par exemple :

```
>>> variance([10, 10, 10, 10])
0.0
```

```
>>> variance([20, 0, 20, 0])
100.0
```

Question 5

L'*écart-type* d'une liste de nombres est égal à la racine carrée de la variance de la liste.

Donner une définition de la fonction `ecart_type` qui, étant donné une liste non vide de nombres, renvoie l'écart-type de la liste.

```
>>> ecart_type([10, 10, 10, 10])
0.0
```

```
>>> ecart_type([20, 0, 20, 0])
10.0
```

```
>>> ecart_type([15, 15, 5, 5])
5.0
```

```
>>> ecart_type([12, 11, 10, 12, 11])
0.7483314773547993
```

Exercice 6.7 : Listes obtenues par multiplication ou division

Dans cet exercice, on résout des problèmes de transformation et de filtrage, ainsi que de combinaison de listes.

Question 1

Donner une définition de la fonction `liste_mult` qui, étant donné une liste L d'entiers et un entier k, retourne la liste obtenue en multipliant par k tous les éléments de L.

Par exemple :

```
>>> list_mult([3, 5, 9, 4], 2)
[6, 10, 18, 8]
```

```
>>> list_mult([], 2)
[]
```

Question 2

Donner une définition de la fonction `liste_div` qui, étant donné une liste L d'entiers et un entier k non nul, retourne la liste obtenue en divisant
par k les éléments de L qui sont multiples de k et en supprimant les autres.

Par exemple :

```
>>> list_div([2, 7, 9, 24, 6], 2)
[1, 12, 3]
```

```
>>> list_div([2, 7, 9, 24, 6], 3)
[3, 8, 2]
```

```
>>> list_div([2, 7, 9, 24, 6], 5)
[]
```

```
>>> list_div([2, 7, 9, -24, 6], -3)
[-3, 8, -2]
```

```
>>> list_div([], 3)
[]
```

Exercice 6.8 : Entrelacement de deux listes

Cet exercice propose de résoudre des problèmes d'entrelacement, qui sortent donc du cadre des problèmes de réduction, transformation ou filtrage.

Question 1

Donner une définition de la fonction `entrelacement` qui, étant donné deux listes `l1` et `l2` de même type et de même longueur, renvoie la liste obtenue en intercalant
les éléments de `l1` et ceux de `l2` : le premier élément de `l1` puis le premier élément de `l2` puis le deuxième élément de `l1` puis le deuxième élément de `l2`, etc.

Par exemple :

```
>>> entrelacement([1, 2, 3], [4, 5, 6])
[1, 4, 2, 5, 3, 6]
```

Question 2

Donner une définition de la fonction `entrelacement_general` qui, étant donné deux listes `l1` et `l2` de même type, renvoie la liste obtenue en intercalant
les éléments de `l1` et ceux de `l2`. Si l'une des listes est plus longue que l'autre, on ajoute les éléments restants en fin de la liste résultat.

Par exemple :

```
>>> entrelacement_general([1,2,3],[4,5,6])
[1, 4, 2, 5, 3, 6]
```

```
>>> entrelacement_general([1,2,3],[4,5,6,7,8])
[1, 4, 2, 5, 3, 6, 7, 8]
```

```
>>> entrelacement_general([1,2,3,4,5],[6,7,8])
[1, 6, 2, 7, 3, 8, 4, 5]
```

```
def entrelacement_general(l1 : List[T], l2 : List[T]) -> List[T]:
    """Retourne la liste obtenue en intercalant les éléments de l1 et ceux de l2.
    """
```

```python
    # liste résultat
    lres : List[T] = []

    i : int # indice courant
    for i in range(max(len(l1), len(l2))):
        if i < len(l1):
            lres.append(l1[i])
        if i < len(l2):
            lres.append(l2[i])
    return lres
```

Une autre solution qui utilise `entrelacement` :

Exercice 6.9 : Conversions de chaînes en listes et vice-versa.

Dans cet exercice, nous nous intéressons aux conversions de chaînes de caractères vers des listes et *vice-versa*.

Question 1 : Jonction

Dans cette question nous proposons de convertir une liste de chaînes de caractères vers une chaîne de caractère.

Le principe est simplement d'effectuer une jonction (ou concaténation) des chaînes de la liste en les séparant avec un caractère séparateur.

La spécification de la fonction est la suivante :

```python
def jonction(l : List[str], c : str) -> str:
    """Précondition : len(c) = 1
    Retourne la chaîne composée de la jonction des
    chaîne de L séparées deux-à-deux par le
    caractère séparateur c."""
```

Par exemple :

```python
>>> jonction(['un', 'deux', 'trois', 'quatre'], '.')
'un.deux.trois.quatre'
```

```python
>>> jonction(['les', 'mots', 'de', 'cette', 'phrase'], ' ')
'les mots de cette phrase'
```

```python
>>> jonction(['un'], '+')
'un'
```

```python
>>> jonction([], '+')
''
```

Donner une définition de la fonction `jonction` spécifiée ci-dessus.

Question 2 : séparation

Dans cette question, nous souhaitons convertir une chaîne de caractères en une liste de chaînes de caractères selon la spécification suivante :

```
def separation(s : str, c :str) -> List[str]:
    """Précondition : len(c) = 1

    retourne la liste de chaînes composées des sous-chaînes
    de s séparées par le caractère séparateur c.
    Le séparateur c n'est pas présent dans la chaîne résultat."""
```

Par exemple :

```
>>> separation('un.deux.trois.quatre', '.')
['un', 'deux', 'trois', 'quatre']
```

```
>>> separation('les mots de cette phrase', ' ')
['les', 'mots', 'de', 'cette', 'phrase']
```

```
>>> separation('les mots de cette phrase', '.')
['les mots de cette phrase']
```

```
>>> separation('', '+')
[]
```

Donner une définition de la fonction `separation` spécifiée ci-dessus.

Thème 7 : Exercices avancés sur les listes

Exercice 7.1 : Nombres complexes (corrigé)

Le but de cet exercice est de définir quelques opérations simples sur les nombres complexes. Il existe un type `complex` prédéfini en Python, mais nous allons manipuler les nombres complexes sous la forme de couples de flottants dans cet exercice.

Question 1

Nous commençons par définir un alias de type pour les nombres complexes.

```
Complexe = Tuple[float, float]
```

Ainsi le nombre complexe 2 + 3i sera représenté par le couple $(2.0, 3.0)$, le nombre i par $(0.0, 1.0)$ et un réel r par $(r, 0.0)$.

Donner la spécification et une définition en Python des fonctions `partie_relle` et `partie_imaginaire` telles que `partie_reelle(c)` (resp. `partie_imaginaire(c)`) renvoie la partie réelle (resp. imaginaire) d'un nombre complexe. Par exemple :

```
>>> partie_reelle((2.0,3.0))
2.0
```

```
>>> partie_imaginaire((2.0,3.0))
3.0
```

```
>>> partie_reelle((0.0,1.0))
0.0
```

```
>>> partie_imaginaire((0.0,1.0))
1.0
```

```
>>> partie_reelle((4.0,0.0))
4.0
```

```
>>> partie_imaginaire((4.0,0.0))
0.0
```

Question 2

Donner la spécification et une définition en Python de la fonction `addition_complexe` telle que `addition_complexe(c1, c2)` renvoie l'addition des nombres complexes `c1` et `c2`

Par exemple :

```
>>> addition_complexe((1.0, 0.0), (0.0, 1.0))
(1.0, 1.0)
```

```
>>> addition_complexe((2.0, 3.0), (0.0, 1.0))
(2.0, 4.0)
```

Question 3

On rappelle que le produit de deux nombres complexes $(a + bi)$ et $(c + di)$ est donné par

$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$. Donner la spécification et une définition en Python de la fonction `produit_complexe` telle que `produit_complexe(c1, c2)` renvoie le produit des nombres complexes `c1` et `c2`

Par exemple :

```
>>> produit_complexe((0.0, 0.0), (1.0, 1.0))
(0.0, 0.0)
```

```
>>> produit_complexe((0.0, 1.0), (0.0, 1.0))
(-1.0, 0.0)
```

```
>>> produit_complexe((2.0, 3.0), (0.0, 1.0))
(-3.0, 2.0)
```

Exercice 7.2 : Nombre d'occurrences du maximum (corrigé)

Cet exercice montre l'utilisation de n-uplet pour améliorer l'efficacité de la résolution d'un problème.

Question

Dans l'exercice 6.2 (cf. chapitre 6), nous avons écrit une fonction (`nb_max`) permettant de calculer le nombre de fois que le maximum d'une liste apparaît dans cette liste. La solution proposée consistait à parcourir une première fois la liste pour déterminer le maximum, puis une seconde fois pour compter le nombre d'occurrences de ce maximum.

Afin d'améliorer l'efficacité de cette fonction, on aimerait ne parcourir qu'une seule fois la liste afin de déterminer *à la fois* le maximum et le nombre d'occurrences de ce dernier.

Donner la spécification et une définition en Python de la fonction `nb_de_max` qui, étant donné une liste non vide de nombres, renvoie un couple contenant le maximum et le nombre de fois où ce maximum apparaît dans la liste. Par exemple

```
>>> nb_de_max([10])
(10, 1)
```

```
>>> nb_de_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
(9, 2)
```

```
>>> nb_de_max([-2, -1, -5, -3, -1, -4, -1])
(-1, 3)
```

Exercice 7.3 : Fichiers texte et système de facturation (sur machine, corrigé)

Cet exercice, à effectuer sur machine, fournit les bases du traitement de fichiers texte en Python.

Complément : chargement d'un fichier texte en Python Les fonctions suivantes permettent de charger (resp. sauvegarder) un fichier texte en Python, sous la forme d'une liste de lignes.

```python
def chargement_fichier(nom_fichier : str) -> List[str]:
    """Précondition : nom_fichier est le nom d'un fichier texte existant
    Retourne le contenu du fichier texte identifié par nom_fichier
    sous la forme d'une liste de lignes de texte.
    """
    # Lignes contenues dans le fichier
    l1 : list[str] = []

    # Lignes sans retour charriot
    l2 : list[str] = []

    with open(nom_fichier, 'r') as f:  # 'r' pour read (lecture)
        l1 = f.readlines() # opération de lecture de lignes

    # suppression des retours charriots
    ligne : str
    for ligne in l1:
        if ligne != '' and ligne[-1] == '\n':
            l2.append(ligne[:-1])
        else:
            l2.append(ligne)

    return l2
```

Remarque : la fonction `chargement_fichier` est un petit peu complexe du fait de l'élimination des retours charriot (qui n'est pas faite automatiquement en Python).

On fournit maintenant une fonction complémentaire permettant de sauvegarder un fichier.

```python
def sauvegarde_fichier(nom_fichier : str, Contenu : List[str]) -> None:
    """Précondition : nom_fichier est un nom correct de fichier
    Sauvegarde le Contenu comme lignes de texte dans le
    fichier identifié par nom_fichier
    Attention : si le fichier existe déjà son contenu sera
    effacé.
    """
    with open(nom_fichier, 'w') as f: # 'w' pour write (écriture)
        ligne : str
        for ligne in Contenu:
            f.write(ligne) # écriture de la ligne
            f.write('\n')  # ajout d'un retour charriot


    return None
```

Remarque : dans ce livre, nous ne présentons pas les nombreuses fonctionnalités de la bibliothèque standard de Python, on consultera pour cela le manuel de Python ou un ouvrage plus directement focalisé sur le langage.

Question 1

Après avoir saisi ces fonctions, donner une expression Python permettant de sauvegarder un fichier de nom `haiku.txt` et contenant le texte suivant:

```
Papillon voltige
Dans un monde
Sans espoir.
(Kobayashi Issa)
```

Saisir ensuite une expression pour lire ce fichier sous la forme d'une liste de chaînes de caractères.

Question 2

Donner une définition de la fonction `decoupage_mots` qui, étant donnée une chaîne de caractères `phrase` composée de mots séparés par des espaces, renvoie la liste correspondante des mots de la phrase.

Par exemple :

```
>>> decoupage_mots("Dans un monde")
['Dans', 'un', 'monde']
```

```
>>> decoupage_mots("Bonjour    Hello     ")
['Bonjour', 'Hello']
```

```
>>> decoupage_mots("Unique")
['Unique']
```

```
>>> decoupage_mots("")
[]
```

Question 3

Le gérant d'une petite surface commerciale vous demande de réaliser un petit logiciel permettant de générer des factures à partir de commandes client.

Pour commencer on souhaite créer une facture exemple sous la forme d'un fichier texte `commande.txt` contenant les lignes suivantes :

```
Lait   12   2.0
Thé     8   4.5
Tomate 6   1.5
Fromage 9  8.5
```

Chaque ligne de la commande contient trois informations (des "mots") :

— le nom d'un produit: `Lait`, `Thé`, etc.
— une quantité commandée (un entier)
— un prix unitaire H.T. (un flottant en euros)

On peut créer le fichier `commande.txt` avec un éditeur de texte, ou plus directement en Python :

```
>>> sauvegarde_fichier("commande.txt", ["Lait  12   2.0"
                                       , "Thé    8   4.5"
                                       , "Tomate 6   1.5"
                                       , "Fromage 9  8.5"])
```

Donner la définition d'une fonction `lecture_produit` qui à partir d'une chaîne de caractères contenant une ligne de commande retourne un triplet de type `Tuple[str, int, float]` représentant les trois informations décrites ci-dessus.

Par exemple :

```
>>> lecture_produit("Lait  12 2.0")
('Lait', 12, 2.0)
```

```
>>> lecture_produit("Tomate 6   1.5")
('Tomate', 6, 1.5)
```

Remarque : on pourra utiliser la fonction `decoupage_mots` de la question précédente, ainsi que la fonction primitive `int` (resp. `float`) permettant de convertir une chaîne de caractères en entier (resp. flottant).

```
>>> int("12")
12
```

```
>>> float("2.0")
2.0
```

Question 4

Donner une définition de la fonction `lecture_commande` qui à partir d'une `commande` composée d'une liste de lignes de produits commandés (cf. question 3), renvoie la liste des triplets de produit correspondant.

Par exemple :

```
>>> lecture_commande(["Lait  12   2.0"
                     , "Thé    8   4.5"
                     , "Tomate 6   1.5"
                     , "Fromage 9  8.5"])
[('Lait', 12, 2.0), ('Thé', 8, 4.5), ('Tomate', 6, 1.5), ('Fromage', 9, 8.5)]
```

Donner ensuite une expression permettant de générer la liste des produits directement à partir du fichier texte `commande.txt`.

Voici l'expression cherchée :

Question 5

Le gérant du magasin nous demande pour finir d'éditer, sous la forme d'un fichier texte une facture correspondant à une commande. La facture pour notre commande exemple (fichier `commande.txt`) sera sauvegardée sous la forme d'un fichier `facture.txt` et composée de la façon suivante :

```
Produit  Prix
-------  -----
Lait     24.0
Thé      36.0
Tomate   9.0
Fromage  76.5

Total_HT   145.5
```

```
TVA_20%    29.1
Total_TTC  174.6
```

Pour cela, on donnera tout d'abord une définition de la fonction `gen_facture` qui, étant donnée une liste de triplets produits, renvoie une liste de lignes de texte correspondant à la facture.

Par exemple :

```
>>> gen_facture([('Lait', 12, 2.0)
                 , ('Thé', 8, 4.5)
                 , ('Tomate', 6, 1.5)
                 , ('Fromage', 9, 8.5)])
['Produit  Prix',
 '-------  ----',
 'Lait  24.0',
 'Thé  36.0',
 'Tomate  9.0',
 'Fromage  76.5',
 '',
 'Total_HT    145.5',
 'TVA_20%     29.1',
 'Total_TTC   174.6']
```

Donner finalement une expression permettant de générer le fichier `facture.txt` à partir du fichier `commande.txt`.

Vous pourrez ensuite modifier la commande pour voir les changements sur la facturation. Le système informatique de votre magasin est prêt à l'emploi !

L'expression cherchée est la suivante :

Exercice 7.4 : Fractions

Dans cet exercice, nous manipulons des *fractions* rationnelles représentées par le type :

```
Tuple[int, int]
```

Par exemple, la fraction $\frac{2}{3}$ est représentée par le couple `(2, 3)` en Python.

Question 1

Une *fraction* $\frac{a}{b}$ où a et b sont des entiers ($b \neq 0$) représente un nombre rationnel. Un inconvénient de cette représentation est qu'un même nombre rationnel peut-être représenté par une infinité de fractions.

Par exemple : le rationnel 1.5 peut être représenté par la fraction $\frac{3}{2}$ mais également la fraction $\frac{6}{4}$, $\frac{30}{20}$, etc. La fraction $\frac{3}{2}$ est appelée la *fraction canonique* (ou *irréductible*) de 1.5.

A partir d'une fraction quelconque $\frac{a}{b}$ (avec $b \neq 0$), la fraction canonique correspondante est :

$$\frac{a/p}{b/p} \text{ avec } p = pgcd(a, b)$$

Par exemple : la fraction canonique de $\frac{9}{12}$ est :

$\frac{9/3}{12/3} = \frac{3}{4}$ avec $pgcd(9, 12) = 3$.

Définir une fonction `fraction` qui, étant donné deux entiers `a` et `b` avec `b` non nul, retourne la fraction canonique de $\frac{a}{b}$.

Par exemple :

```
>>> fraction(9, 12)
(3, 4)
```

```
>>> fraction(12, 9)
(4, 3)
```

```
>>> fraction(180, 240)
(3, 4)
```

```
>>> fraction(121, 187)
(11, 17)
```

Rappel : le cours 3 propose une fonction du calcul du pgcd de deux entiers `a` et `b`.

Question 2

Proposer une définition de la fonction `frac_mult` qui retourne la multiplication de deux fractions `f1` et `f2` sous la forme d'une fraction canonique.

Par exemple :

```
>>> frac_mult( (3, 4), (8, 4) )
(3, 2)
```

```
>>> frac_mult( (3, 4), (4, 3) )
(1, 1)
```

```
>>> frac_mult( (3, 4), (1, 1) )
(3, 4)
```

```
>>> frac_mult( (3, 4), (0, 2) )
(0, 1)
```

Question 3

En utilisant `frac_mult`, proposer une définition de la fonction `frac_div` de division entre deux fractions rationnelles.

Question 4

La somme de deux fractions est obtenue par la formule suivante :

$$\frac{a}{b} + \frac{c}{d} = \frac{a \times (p/b) + c \times (p/d)}{p}$$

avec $p = ppcm(b, d)$

Pour le calcul du `ppcm` on utilise la fonction suivante :

```python
def ppcm(a : int, b : int) -> int:
    """Précondition : (a != 0) and (b != 0)

    Retourne le plus petit commun multiple de a et b.
    """

    # pgcd de a et b
    p : int = 0
    if a >= b:
        p = pgcd(a, b)
    else:
        p = pgcd(b, a)

    return abs(a * b) // p
```

```python
# Jeu de tests
assert ppcm(3, 4) == 12
assert ppcm(4, 3) == 12
assert ppcm(11, 17) == 187
assert ppcm(15, 9) == 45
```

Proposer une définition de la fonction `frac_add` qui retourne la fraction canonique correspondant à la somme de deux fractions `f1` et `f2`.

Par exemple :

```python
>>> frac_add( (8, 4), (1, 4) )
(9, 4)
```

```python
>>> frac_add( (9, 4), (5, 4) )
(7, 2)
```

```python
>>> frac_add( (1, 3), (1, 2) )
(5, 6)
```

Exercice 7.5 : Tester l'alignement de points

Le but de cet exercice est de vérifier qu'une liste de points ne contient que des points alignés.

Le type Point sera défini dans tout l'exercice par un couple d'entiers. On choisit délibérément de travailler sur des entiers, afin d'éviter des problèmes d'arrondis. On définit donc formellement l'alias de type suivant :

```python
Point = Tuple[int, int]
```

que l'on pourra utiliser dans les signatures de fonctions.

Question 1

Donner la spécification et une définition en Python de la fonction `vecteur` telle que `vecteur(p1, p2)` renvoie le couple de coordonnées du vecteur formé par les points `p1` et `p2`.

On rappelle que les coordonnées du vecteur correspond au couple (différence des abscisses, différence des ordonnées).

Question 2

Donner la spécification et une définition en Python de la fonction `alignes` telle que `alignes(p1, p2, p3)` renvoie le booléen `True` si les 3 points sont alignés et `False` sinon.

On rappelle que 3 points $p1, p2, p3$ sont alignés si les deux vecteurs $p1 \cdot p2$ et $p2 \cdot p3$ sont colinéaires, c'est-à-dire proportionnels.

Par exemple :

```
>>> alignes((0,0), (1,1), (5,5))
True
```

```
>>> alignes((0,0), (1,1), (1,2))
False
```

Question 3

Donner la spécification et une définition en Python de la fonction `alignement` qui, étant donné une liste `L` de points contenant au moins 3 éléments, renvoie le booléen `True` si tous les points de la liste `L` sont alignés et `False` sinon.

Par exemple :

```
>>> alignement([(0,0), (1,1), (5,5)])
True
```

```
>>> alignement([(0,0), (1,1), (5,5), (1,0)])
False
```

Exercice 7.6 : Base de données des étudiants

Dans cet exercice, nous illustrons une utilisation courante des listes de n-uplets : la manipulation d'une base de données.

Nous manipulerons une base de données composée d'une liste d'enregistrements, chaque enregistrement étant un quadruplet avec :

1. le nom de l'étudiant de type `str`
2. le prénom de l'étudiant de type `str`
3. son numéro d'étudiant de type `int`
4. d'une liste de notes sur 20 obtenues aux examens, de type `List[int]`

On définit donc *l'alias de type* suivant :

```
Etudiant = Tuple[str, str, int, List[int]]
```

On fait de plus l'hypothèse implicite que toutes les notes enregistrées sont entre 0 et 20.

La base de données manipulée est donc du type `List[Etudiant]`.

Pour l'exercice on considérera la base de données suivante :

```python
BaseUPMC : List[Etudiant]
BaseUPMC = [('GARGA', 'Amel', 20231343, [12, 8, 11, 17, 9]),
            ('POLO', 'Marcello', 20342241, [9, 11, 19, 3]),
            ('AMANGEAI', 'Hildegard', 20244229, [15, 11, 7, 14, 12]),
            ('DENT', 'Arthur', 42424242, [8, 4, 9, 4, 12, 5]),
            ('ALEZE', 'Blaise', 30012024, [17, 15, 20, 14, 18, 16, 20]),
            ('D2', 'R2', 10100101, [10, 10, 10, 10, 10, 10])]
```

Question 1 : moyenne des notes

Donner une définition de la fonction `note_moyenne` qui, à partir d'une liste de notes (entre 0 et 20) retourne leur moyenne.

Par exemple :

```python
>>> note_moyenne([12, 8, 14, 6, 5, 15])
10.0
```

```python
>>> note_moyenne([])
0.0
```

Question 2 : moyenne générale

Donner une définition de la fonction `moyenne_generale` qui, étant donné une base de données d'étudiants, retourne la moyenne générale des notes des étudiants enregistrés (c'est-à-dire la moyenne des moyennes de chaque étudiant).

Par exemple :

```python
>>> moyenne_generale(BaseUPMC)
11.307142857142857
```

```python
>>> moyenne_generale([])
0.0
```

Question 3 : Nom et prénom du meilleur étudiant

On cherche maintenant dans la base le nom et le prénom d'un étudiant qui possède la meilleure moyenne. La spécification utilisée est la suivante :

```python
def top_etudiant(bd : List[Etudiant]) -> Tuple[str, str]
    """Précondition : len(bd) > 0
    retourne un étudiant de la base bd avec la meilleure
    moyenne. Si des étudiants sont ex-aequo alors on
    retourne le premier dans l'ordre séquentiel de la liste."""
```

Pour notre base UPMC on obtient :

```python
>>> top_etudiant(BaseUPMC)
('ALEZE', 'Blaise')
```

Question 4 : Recherche d'une moyenne

Donner une définition de la fonction partielle `recherche_moyenne` qui étant donné un numéro d'étudiant `rnum` ainsi qu'une base de données `bd`, retourne la moyenne de l'étudiant correspond ou `None` si ce numéro d'étudiant est inconnu.

Par exemple :

```
>>> recherche_moyenne(20244229, BaseUPMC)
11.8
```

```
>>> recherche_moyenne(20342241, BaseUPMC)
10.5
```

```
>>> recherche_moyenne(2024129111, BaseUPMC)
```

Remarque : dans ce dernier cas, `None` est retourné et donc l'interprète Python ne montre pas de réponse.

Exercice 7.7 : Intersection de listes

Le but de cet exercice est d'écrire une fonction permettant de calculer l'intersection d'une liste de listes d'entiers, chacune triée dans l'ordre croissant.

Question 1

Donner la spécification et une définition en Python de la fonction `intersection_2_listes` qui, étant donné deux listes d'entiers `l1` et `l2` triées en ordre croissant, renvoie la liste des éléments appartenant à la fois à `l1` et à `l2`. On exploitera bien sûr le fait que `l1` et `l2` sont toutes deux triées. Par exemple

```
>>> intersection_2_listes([0,1,2], [3,4,5])
[]
```

```
>>> intersection_2_listes([1,2,3],[1,2,3])
[1, 2, 3]
```

```
>>> intersection_2_listes([1,1],[1,1])
[1, 1]
```

```
>>> intersection_2_listes([1,1],[1,2])
[1]
```

```
>>> intersection_2_listes([],[1,2,3])
[]
```

```
>>> intersection_2_listes([1,2,2,3,4],[2,3,4,4,5,6])
[2, 3, 4]
```

Question 2

Soit `l` une liste de listes d'entiers, chacune triée en ordre croissant. Donner la spécification et une définition en Python de la fonction `intersection` telle que `intersection(l)` renvoie la liste triée des entiers appartenant à chacune des listes de `l`.

```
intersection([[1, 2, 3, 4, 4, 5], [2, 5, 7], [0, 2, 2, 4, 4, 5, 9]])
[2, 5]
```

```
intersection([[1, 2, 3, 4, 4, 5], [2, 4, 4, 5, 7], [0, 2, 2, 4, 4, 5, 9]])
[2, 4, 4, 5]
```

Exercice 7.8 : Carrés magiques

L'objectif de cet exercice est de vérifier les contraintes des carrés magiques représentés par des listes de listes d'entiers. Par la même occasion, on illustre ici la décomposition d'un problème (relativement) complexe en un certain nombre de sous-problèmes plus simples.

Un carré magique de dimension n est une matrice de taille $n \times n$ contenant des entiers naturels, tel que :

- tous les entiers de 1 à $n \times n$ sont présents dans la matrice (donc tous les éléments sont distincts),
- les sommes des entiers de chaque ligne sont toutes égales à une même valeur S,
- la somme de chaque colonne est également S,
- les sommes des deux diagonales de la matrice sont également S.

Voici un exemple de carré magique de dimension 3 :

2	7	6
9	5	1
4	3	8

On a bien ici :

- tous les entiers de 1 à $3 \times 3 = 9$ sont présents
- la somme de chaque ligne est $2 + 7 + 6 = 9 + 5 + 1 = 4 + 3 + 8 = 15$
- la somme de chaque colonne est $2 + 9 + 4 = 7 + 5 + 3 = 6 + 1 + 8 = 15$
- la somme de chaque diagonale est $2 + 5 + 8 = 6 + 5 + 4 = 15$

En Python, nous allons représenter le carré magique en utilisant le type `list[list[int]]`. Pour un carré magique de taille n, chaque ligne de la matrice est représentée par une liste d'entiers. Les lignes sont elle-mêmes stockées dans une liste dans l'ordre séquentiel (la première liste correspond à la première ligne, la seconde liste correspond à la deuxième ligne, etc.).

Voici une expression permettant de construire le carré magique ci-dessus :

```
[ [2, 7, 6],
  [9, 5, 1],
  [4, 3, 8] ]
```

- les lignes sont respectivement :
  [2, 7, 6] et [9, 5, 1] et [4, 3, 8]
- les colonnes sont respectivement :
  [2, 9, 4] et [7, 5, 3] et [6, 1, 8]
- les diagonales sont :
  [2, 5, 8] et [6, 5, 4]

Pour simplifier les tests par la suite, on associe ce carré magique à une variable :

```python
CarreMagique : List[List[int]]
CarreMagique = [ [2, 7, 6],
                 [9, 5, 1],
                 [4, 3, 8] ]
```

Question 1

Donner une définition du prédicat `presence` qui étant donné un entier `n` et une liste d'entiers `l` retourne `True` si l'entier `n` est présent dans la liste `l` ou `False` sinon.

Par exemple :

```python
>>> presence(5, [9, 5, 1])
True
```

```python
>>> presence(4, [9, 5, 1])
False
```

Question 2

Donner une définition du prédicat `mat_presence` qui étant donné un entier `n` et une liste de listes d'entiers `ll` retourne `True` si l'entier `n` est présent dans la liste `ll` ou `False` sinon.

Par exemple :

```python
>>> mat_presence(5, [[1, 2, 3], [4, 5, 6]])
True
```

```python
>>> mat_presence(7, [ [1,2, 3], [4, 5, 6] ])
False
```

```python
>>> mat_presence(7, CarreMagique)
True
```

```python
>>> mat_presence(10, CarreMagique)
False
```

Question 3

Donner une définition du prédicat `verif_elems` qui, étant donné un entier naturel `n` non nul et une liste `ll` de listes d'entiers, retourne `True` si tous les entiers dans l'intervalle $[1; n \times n]$ sont présents dans la liste `ll`, ou `False` sinon.

Par exemple :

```python
>>> verif_elems(3, CarreMagique)
True
```

```python
>>> verif_elems(3, [ [2, 7, 6], [8, 5, 1], [4, 3, 8] ])
False
```

Question 4

Donner une définition de la fonction `somme_liste` qui retourne la somme des éléments d'une liste d'entiers.

Par exemple :

```
>>> somme_liste([2, 7, 6])
15
```

```
>>> somme_liste([9, 5, 1])
15
```

```
>>> somme_liste([4, 3, 8])
15
```

Question 5

Donner une définition du prédicat `verif_lignes` tel que `verif_lignes(ll, s)` retourne `True` si toutes les sous-listes de `ll` possèdent la même somme `s`, ou `False` sinon.

Par exemple :

```
>>> verif_lignes(CarreMagique, 15)
True
```

```
>>> verif_lignes(CarreMagique, 16)
False
```

```
>>> verif_lignes([ [2, 7, 6], [8, 5, 1], [4, 3, 8] ], 15)
False
```

Question 6

Donner une définition de la fonction `colonne` qui, étant donné un entier `j` et une matrice `mat` de dimension n, retourne la `j`-ième colonne de la matrice `mat`. On fait l'hypothèse que `j` est compris entre 0 et n.

Par hypothèse, une matrice de dimension n est une liste de n listes d'entiers où chaque sous-liste est de longueur n.

Par exemple :

```
>>> colonne(0, [ [2, 7, 6],
                 [9, 5, 1],
                 [4, 3, 8] ])
[2, 9, 4]
```

```
>>> colonne(1, [ [2, 7, 6],
                 [9, 5, 1],
                 [4, 3, 8] ])
[7, 5, 3]
```

```
>>> colonne(2, [ [2, 7, 6],
                 [9, 5, 1],
                 [4, 3, 8] ])
[6, 1, 8]
```

Question 7

Donner une définition du prédicat `verif_colonnes` tel que `verif_colonnes(s, mat)` retourne `True` si toutes les colonnes de la matrice `mat` possèdent la même somme `s`, ou `False` sinon.

Par exemple :

```
>>> verif_colonnes(CarreMagique, 14)
False
```

```
>>> verif_colonnes(CarreMagique, 15)
True
```

```
>>> verif_colonnes([ [2, 7, 6], [8, 5, 1], [4, 3, 8] ], 15)
False
```

Question 8

Donner une définition de la fonction `diagonale_1` (resp. `diagonale_2`) qui, étant donné une matrice `mat` de dimension n, retourne la liste formée des éléments de la première diagonale (resp. seconde diagonale).

Par exemple :

```
>>> diagonale_1([ [2, 7, 6],
                  [9, 5, 1],
                  [4, 3, 8] ])
[2, 5, 8]
```

```
>>> diagonale_2([ [2, 7, 6],
                  [9, 5, 1],
                  [4, 3, 8] ])
[6, 5, 4]
```

```
>>> diagonale_1([ [ 4, 14, 15,  1],
                  [ 9,  7,  6, 12],
                  [ 5, 11, 10,  8],
                  [16,  2,  3, 13] ])
[4, 7, 10, 13]
```

```
>>> diagonale_2([ [ 4, 14, 15,  1],
                  [ 9,  7,  6, 12],
                  [ 5, 11, 10,  8],
                  [16,  2,  3, 13] ])
[1, 6, 11, 16]
```

Question 9

Donner (finalement !) une définition du prédicat `verif_magique` qui, étant donné une matrice `mat` de dimension n, retourne `True` si et seulement si elle représente un carré magique.

Par exemple :

```
>>> verif_magique(CarreMagique)
True
```

```
>>> verif_magique([ [2, 7, 6], [8, 5, 1], [4, 3, 8] ])
False
```

Thème 8 : Exercices sur les compréhensions de listes

Exercice 8.1 : Revisiter les listes (corrigé)

L'objectif de cet exercice consiste à revisiter certains exercices sur les listes du chapitre 6 et de proposer de nouvelles solutions basées sur les expressions de compréhension.

Question 1

Proposer une solution basée sur les compréhensions pour la question 1 de l'exercice 6.1 *Listes de répétitions* du chapitre 6.

Question 2

Proposer une solution basée sur les compréhensions pour les questions 1 et 2 de l'exercice 6.4 *Liste de diviseurs* du chapitre 6.

Exercice 8.2 : Lettres de l'alphabet (corrigé)

Dans cet exercice, nous manipulons des chaînes de caractères et des listes en utilisant des compréhensions.

Question 1

Dans l'exercice *Conversions* du thème 5 sur les chaînes de caractères, nous avons utilisé les primitives `ord` et `chr` de Python qui permettent de convertir un caractère en son numéro Unicode (entier) et *vice-versa*.

Par exemple :

```
>>> ord('a')
97
```

Le numéro Unicode du caractère `'a'` est l'entier `97`

```
>>> ord('z')
122
```

Et les numéros vont croissant jusqu'au numéro `122` pour le caractère `'z'`.

On peut vérifier que l'on dispose bien de 26 numéros pour l'alphabet latin :

```
>>> ord('z') - ord('a') + 1
26
```

Pour repasser d'un numéro Unicode à un caractère, on utilise la primitive `chr` :

```
>>> chr(97)
'a'
```

```
>>> chr(122)
'z'
```

```
>>> chr(103)
'g'
```

En utilisant ces deux fonctions, proposer une définition de la fonction `alphabet` sans paramètre et qui construit la liste des lettres de l'alphabet à l'aide d'une compréhension.

Ainsi, le seul jeu de tests intéressant pour cette fonction est le suivant :

```
# Jeu des tests
assert alphabet() == ['a','b','c','d','e','f','g','h','i','j','k','l','m',\
                      'n','o','p','q','r','s','t','u','v','w','x','y','z']
```

Question 2

Donner une définition de la fonction `est_voyelle` qui étant donné un caractère `c`, retourne `True` si et seulement si `c` est une voyelle de l'alphabet à 26 lettres.

Question 3

Donner une expression de compréhension permettant d'obtenir la liste des voyelles de l'alphabet.

Question 4

Même question pour les consonnes de l'alphabet.

Exercice 8.3 : Crible d'Eratosthène (corrigé)

Cet exercice consiste à implémenter le crible d'Eratosthène qui décrit un moyen de calculer la listes des nombres premiers inférieurs à un entier fixé. On rappelle qu'un nombre est premier s'il n'est divisible que par 1 et lui-même. On rappelle également que, par convention, 1 n'est pas un nombre premier.

La méthode du crible d'Eratosthène consiste à créer dans un premier temps la liste des entiers compris entre 2 et n, puis itérer le processus suivant :

1. Récupérer le premier élément de la liste (ce nombre est un nombre premier)
2. Retirer de la liste restante tous les multiples de ce nombre.
3. Recommencer à l'étape 1. tant qu'il reste des éléments dans la liste

Illustrons la méthode avec l'entier $n = 10$. On commence par créer la liste

`[2, 3, 4, 5, 6, 7, 8, 9, 10]`

On récupère alors le prochain nombre premier, `2`, et on calcule la liste dans laquelle les multiples de 2 ont été retirés, soit `[3, 5, 7, 9]`.

On récupère alors le prochain nombre premier, `3`, et on calcule la liste dans laquelle les multiples de 3 ont été retirés, soit `[5, 7]`.

On récupère alors le prochain nombre premier, `5`, et on calcule la liste dans laquelle les multiples de 5 ont été retirés, soit `[7]`.

On récupère alors le prochain nombre premier, `7`, et on calcule la liste dans laquelle les multiples de 7 ont été retirés, soit `[]`.

La liste est vide, donc le calcul est terminé. La liste des nombres premiers inférieurs ou égaux à 10 est donc `[2, 3, 5, 7]`.

Question 1

À l'aide d'une compréhension, donner la définition et la spécification de la fonction `liste_non_multiple` qui, étant donné un entier `n` non nul et une liste d'entiers `l`, renvoie la liste des éléments de `l` qui ne sont pas multiples de `n`. Par exemple :

```
>>> liste_non_multiple(2,[2,3,4,5,6,7,8,9,10])
[3, 5, 7, 9]
```

```
>>> liste_non_multiple(3,[2,3,4,5])
[2, 4, 5]
```

```
>>> liste_non_multiple(2,[2,4,6])
[]
```

```
>>> liste_non_multiple(2,[])
[]
```

```
>>> liste_non_multiple(7,[2,3,4,5])
[2, 3, 4, 5]
```

Question 2

Donner la définition et la spécification de la fonction `eratosthene` qui calcule la liste des nombres premiers inférieurs ou égaux à un entier `n` (supérieur ou égal à 2) en utilisant le crible d'Eratosthène décrit ci-dessus. Par exemple :

```
>>> eratosthene(10)
[2, 3, 5, 7]
```

```
>>> eratosthene(2)
[2]
```

```
>>> eratosthene(40)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Question 3

Un facteur premier d'un entier n est un nombre premier qui divise n. À l'aide de la fonction précédente, et en utilisant une compréhension, donner la définition et la spécification de la fonction `liste_facteurs_premiers` qui, étant donné un entier `n` supérieur ou égal à 2, calcule la liste des facteurs premiers de `n`, c'est à dire la liste des nombres premiers inférieurs ou égaux à `n` et qui divisent `n`. Par exemple :

```
>>> liste_facteurs_premiers(2)
[2]
```

```
>>> liste_facteurs_premiers(10)
[2, 5]
```

```
>>> liste_facteurs_premiers(2*3*7)
[2, 3, 7]
```

```
>>> liste_facteurs_premiers(2*3*4*7*9)
[2, 3, 7]
```

Exercice 8.4 : Variance

Cet exercice permet de manipuler des compréhensions simples sur une thématique de statistique.

La notion de moyenne est pratique pour synthétiser une série de nombres en ce qu'elle donne une valeur *caractéristique* de cette série. Cependant des séries de nombres différents peuvent avoir une moyenne identique. Ainsi, les listes `[10,10,10]` et `[0,10,20]` ont toutes deux une moyenne de 10. Dans le premier cas pourtant, la valeur 10 est bien plus représentative que dans le deuxième cas puisque **toutes** les nombres de la liste valent `10`.

La notion de *variance* permet justement de rendre compte de ce phénomène en calculant la dispersion des valeurs de la liste vis-à-vis de la moyenne. Plus précisément, elle calcule la moyenne des carrés des écarts à la moyenne. Ainsi dans notre exemple, la variance de la première liste est :

$$((10 - 10)^2 + (10 - 10)^2 + (10 - 10)^2)/3 = 0/3 = 0$$

Alors que la variance de la seconde liste est :

$$((0 - 10)^2 + (10 - 10)^2 + (20 - 10)^2)/3 = 200/3 = 66.66666666$$

Cet exercice consiste à écrire une fonction permettant de calculer la variance d'une série de nombres. Le but est bien sûr d'utiliser le plus possible les compréhensions pour répondre aux questions. À noter que ce problème du calcul de la variance a déjà été traité lors du thème 6 mais en proposant un autre algorithme que celui présenté ci-dessous.

Question 1

Donner la définition et la spécification de la fonction `moyenne` qui calcule la moyenne d'une liste non vide de nombres. Par exemple :

```
>>> moyenne([10,10,10])
10.0
```

```
>>> moyenne([0,10,20])
10.0
```

```
>>> moyenne([1,2])
1.5
```

Question 2

En utilisant une compréhension, donner la définition et la spécification de la fonction `ecart_nombre` qui, étant donné une liste de nombres `l` et un nombre `x`, renvoie la liste de la valeur absolue de la différence des nombres de `l` avec `x`. Par exemple

```
>>> ecart_nombre([10,10,10],10)
[0, 0, 0]
```

```
>>> ecart_nombre([0,10,20], 10)
[10, 0, 10]
```

```
>>> ecart_nombre([1,2],1.5)
[0.5, 0.5]
```

Question 3

En utilisant une compréhension, donner la définition et la spécification de la fonction `liste_carre` qui, étant donné une liste de nombres `l`, renvoie la liste des carrés des éléments de `l`. Par exemple

```
>>> liste_carre([0,0,0])
[0, 0, 0]
```

```
>>> liste_carre([10,0,10])
[100, 0, 100]
```

```
>>> liste_carre([0.5,0.5])
[0.25, 0.25]
```

Question 4

Donner la définition et la spécification de la fonction `variance` qui, étant donné une liste de nombres `l`, renvoie la variance associée à `l`. Par exemple

```
>>> variance([10,10,10])
0.0
```

```
>>> variance([0,10,20])
66.66666666666667
```

```
>>> variance([1,2])
0.25
```

Question 5

L'implémentation précédente est un peu inefficace car elle parcourt une première fois la liste pour calculer les écarts à la moyenne, puis une seconde fois pour calculer le carré de ces écarts. Proposer une implémentation plus efficace de la fonction `variance` ne faisant qu'un seul parcours.

Dernière variante un peu moins lisible mais plus concise :

```
def variance_ter(l : List[float]) -> float:
    """Retourne la variance associée à l.
    """
    # moyenne des valeurs de L
    m : float = moyenne(l)
    return moyenne([abs(m-e)**2 for e in l])
```

Exercice 8.5 : Codage ROT13

Dans cet exercice, nous étudions un algorithme très simple de codage/décodage de message secret. On aborde à nouveau le passage des chaînes aux listes et *vice-versa*.

Question 1

Donner une définition par compréhension de la fonction `liste_caracteres` qui retourne la liste des caractères d'une chaîne passée en paramètre.

Par exemple :

```
>>> liste_caracteres('les carottes')
['l', 'e', 's', ' ', 'c', 'a', 'r', 'o', 't', 't', 'e', 's']
```

```
>>> liste_caracteres('')
[]
```

Question 2

Donner une définition de la fonction `chaine_de` qui retourne la chaîne de caractères correspondant à la liste de caractères passée en paramètre.

Par exemple :

```
>>> chaine_de(['s','a','l','u','t'])
'salut'
```

```
>>> chaine_de([])
''
```

```
>>> chaine_de(liste_caracteres('les carottes'))
'les carottes'
```

Question 3

Pour pouvoir faire de l'arithmétique de codage sur les caractères, nous allons transformer ces derniers en entiers.

Pour la lettre `'a'` on souhaite utiliser le numéro `0`, pour `'b'` le numéro `1`, ..., et pour `'z'` le numéro `25`.

Donner une définition de la fonction `num_car` qui retourne le numéro du caractère passé en paramètre.

Par exemple :

```
>>> num_car('a')
0
```

```
>>> num_car('b')
1
```

```
>>> num_car('z')
25
```

Remarque : on pourra utiliser la primitive `ord` qui retourne le numéro Unicode d'un caractère (cf. exercice *Lettres de l'alphabet*).

Question 4

Définir la fonction `car_num` qui permet de retrouver le caractère correspondant à un numéro obtenu par `num_car`.

Remarque : pour cette fonction on pourra utiliser `chr` et `ord` (cf. exercice *Lettres de l'alphabet*).

Question 5

L'algorithme de chiffrage ROT13 est une variante du *chiffrage de César* qui exploite le fait qu'il n'y a que 26 lettres dans l'alphabet.

Le codage est trivial :

— la lettre numéro `0` (`'a'`) de l'alphabet est codée par la lettre `0+13=13` (`'n'`)
— la lettre numéro `1` (`'b'`) de l'alphabet est codée par la lettre `1+13=14` (`'o'`)
— la lettre numéro `2` (`'c'`) de l'alphabet est codée par la lettre `2+13=15` (`'p'`)
— ...
— la lettre numéro `11` (`'l'`) de l'alphabet est codée par la lettre `11+13=24` (`'y'`)
— la lettre numéro `12` (`'m'`) de l'alphabet est codée par la lettre `12+13=25` (`'z'`)
— la lettre numéro `13` (`'n'`) de l'alphabet est codée par la lettre `(13+13) % 26=0` (`'a'`)
— la lettre numéro `14` (`'o'`) de l'alphabet est codée par la lettre `(14+13) % 26=1` (`'b'`)
— ...
— la lettre numéro `25` (`'z'`) de l'alphabet est codée par la lettre `(25+13) % 26=12` (`'m'`)

Remarque : pour toute lettre hors alphabet le codage ROT13 retourne l'identité.

Par exemple :

```
>>> rot13('a')
'n'
```

```
>>> rot13('b')
'o'
```

```
>>> rot13('l')
'y'
```

```
>>> rot13('m')
'z'
```

```
>>> rot13('n')
'a'
```

```
>>> rot13('o')
'b'
```

```
>>> rot13('z')
'm'
```

```
>>> rot13('8')
'8'
```

```
>>> rot13(' ')
' '
```

Donner une définition de la fonction `rot13` qui étant donné un caractère `c` retourne son caractère codé selon ROT13.

La propriété fondamentale de la fonction `rot13` est qu'elle est involutive :

> pour toute lettre `l` de l'alphabet : `rot13(rot13(l)) = l`

Cette propriété doit être prise en compte dans le jeu de test de la fonction.

Question 6

Définir la fonction `codage_rot13` permettant de coder un message secret en rot13, en utilisant une compréhension.

Par exemple :

```
>>> codage_rot13('abcdef')
'nopqrs'
```

```
>>> codage_rot13('nopqrs')
'abcdef'
```

```
>>> codage_rot13('les carottes sont cuites')
'yrf pnebggrf fbag phvgrf'
```

```
>>> codage_rot13('yrf pnebggrf fbag phvgrf')
'les carottes sont cuites'
```

```
>>> codage_rot13('nowhere gnat chechen')
'abjurer tang purpura'
```

Question : Que faut-il faire pour *décoder* un message secret ? Justifier avec un exemple.

Pour décoder un message secret, il suffit d'appliquer une deuxième fois la fonction de codage, puisque celle-ci est involutive :

```
secret : str = codage_rot13('les carottes sont cuites')
```

```
>>> secret
'yrf pnebggrf fbag phvgrf'
```

```
>>> codage_rot13(secret)
'les carottes sont cuites'
```

Exercice 8.6 : Base de données compréhensive

Dans cet exercice, nous revisitons avec les compréhensions l'exercice *Base de données des étudiants* du thème 7. Cela permet d'aborder les compréhensions sur les listes de n-uplets.

Nous rappelons la base de données utilisée dans les exemples :

```
Etudiant = Tuple[str, str, int, List[int]]

BaseUPMC : List[Etudiant]
BaseUPMC = [('GARGA', 'Amel', 20231343, [12, 8, 11, 17, 9]),
```

```
             ('POLO', 'Marcello', 20342241, [9, 11, 19, 3]),
             ('AMANGEAI', 'Hildegard', 20244229, [15, 11, 7, 14, 12]),
             ('DENT', 'Arthur', 42424242, [8, 4, 9, 4, 12, 5]),
             ('ALEZE', 'Blaise', 30012024, [17, 15, 20, 14, 18, 16, 20]),
             ('D2', 'R2', 10100101, [10, 10, 10, 10, 10, 10])]
```

Question 1

Donner une définition de la fonction `mauvaise_note` qui étant donné un étudiant `etu`, retourne `True` si `etu` a obtenu au moins une note inférieure à la moyenne, ou `False` sinon.

Question 2

Donner une expression de compréhension permettant de récupérer la liste des étudiants de `BaseUPMC` qui ont au moins une note inférieure à la moyenne.

Votre expression devrait retourner :

```
[('GARGA', 'Amel', 20231343, [12, 8, 11, 17, 9]),
 ('POLO', 'Marcello', 20342241, [9, 11, 19, 3]),
 ('AMANGEAI', 'Hildegard', 20244229, [15, 11, 7, 14, 12]),
 ('DENT', 'Arthur', 42424242, [8, 4, 9, 4, 12, 5])]
```

Question 3

Même question que la précédente, mais on souhaite maintenant obtenir uniquement le nom des étudiants concernés.

On attend donc la valeur suivante :

```
['GARGA', 'POLO', 'AMANGEAI', 'DENT']
```

Question 4

Même question mais on veut cette fois-ci la liste des numéros des étudiants qui n'ont aucune mauvaise note.

Votre réponse doit être la suivante :

```
[30012024, 10100101]
```

Exercice 8.7 : Triplets numériques

Dans cet exercice, nous expérimentons les compréhensions multiples en manipulant des listes de triplets d'entiers.

Question 1

Donner une définition de la fonction `triplets` qui retourne la liste des triplets (i, j, k) sur l'intervalle $[1; n]$ (avec n un entier naturel).

Par exemple :

```
>>> triplets(0)
[]
```

```
>>> triplets(1)
[(1, 1, 1)]
```

```
>>> triplets(2)
[(1, 1, 1),
 (1, 1, 2),
 (1, 2, 1),
 (1, 2, 2),
 (2, 1, 1),
 (2, 1, 2),
 (2, 2, 1),
 (2, 2, 2)]
```

Remarque : votre fonction utilisera une compréhension

Question 2

On souhaite maintenant lister les décompositions sur un intervalle $[1; n]$, c'est-à-dire les triplets (i, j, k) tels que $i + j = k$.

Par exemple :

```
>>> decompositions(0)
[]
```

```
>>> decompositions(1)
[]
```

```
>>> decompositions(2)
[(1, 1, 2)]
```

```
>>> decompositions(3)
[(1, 1, 2), (1, 2, 3), (2, 1, 3)]
```

```
>>> decompositions(4)
[(1, 1, 2), (1, 2, 3), (1, 3, 4), (2, 1, 3), (2, 2, 4), (3, 1, 4)]
```

Question 3

On souhaite maintenant lister les encadrements sur un intervalle $[1; n]$, c'est-à-dire les triplets (i, j, k) tels que $i \leq j \leq k$.

Par exemple :

```
>>> encadrements(0)
[]
```

```
>>> encadrements(1)
[(1, 1, 1)]
```

```
>>> encadrements(2)
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

```
>>> encadrements(3)
[(1, 1, 1),
```

```
(1, 1, 2),
(1, 1, 3),
(1, 2, 2),
(1, 2, 3),
(1, 3, 3),
(2, 2, 2),
(2, 2, 3),
(2, 3, 3),
(3, 3, 3)]
```

Remarque : on essaiera de trouver deux solutions différentes (mais toutes deux basées sur une compréhension multiple) et on discutera de leur efficacité relative.

Question 4 (sur machine)

Proposer deux définitions alternatives de `encadrements` sans utiliser de compréhension.

Sur machine, on pourra compter le nombre d'étapes nécessaires au calcul pour comparer l'efficacité des deux versions.

Solutions des exercices corrigés

Solution de l'exercice 1.1

Solution de la question 1

```python
def moyenne_trois_nb(a : float, b : float, c : float) -> float:
    """Retourne la moyenne arithmétique des trois nombres a, b et c.
    """
    return (a + b + c) / 3.0   # remarque : division flottante
```

```python
# Jeu de tests
assert moyenne_trois_nb(3, 6, -3) == 2.0
assert moyenne_trois_nb(3, 0, -3) == 0.0
assert moyenne_trois_nb(1.5, 2.5, 1.0) == 5.0 / 3.0
assert moyenne_trois_nb(3, 6, 3) == 4.0
assert moyenne_trois_nb(1, 2, 3) == 2.0
assert moyenne_trois_nb(1, 1, 1) == 1.0
assert moyenne_trois_nb(3, 6, 3) == 4.0
assert moyenne_trois_nb(1, 2, 3) == 2.0
```

Solution de la question 2 : moyenne pondérée

```python
def moyenne_ponderee(a : float, b : float, c : float
                     , pa : float, pb : float, pc : float) -> float:
    """Précondition : pa + pb + pc != 0
    Retourne la moyenne des trois nombres a, b, c, pondérés respectivement
    par pa (pondération pour a), pb et pc.
    """
    return ((a * pa) + (b * pb) + (c * pc)) / (pa + pb + pc)
```

```python
# Jeu de tests
assert moyenne_ponderee(1, 1, 1, 3, 6, -3) == 1.0
assert moyenne_ponderee(2, 3, 4, 1, 1, 1) == 3.0
assert moyenne_ponderee(1, 0, 4, 2, 1, 2) == 2.0
```

Solution de l'exercice 1.2

Solution de la question 1

```python
def prix_ttc(prix : float, taux : float) -> float:
    """Précondition : prix >= 0
    Retourne le prix TTC correspondant au prix HT 'prix'
    avec un taux de TVA 'taux'.
    """
    return prix * (1 + taux / 100.0)
```

```python
# Jeu de tests
assert prix_ttc(100.0, 20.0) == 120.0
```

```python
assert prix_ttc(100, 0.0) == 100.0
assert prix_ttc(100, 100.0) == 200.0
assert prix_ttc(0, 20) == 0.0
assert prix_ttc(200, 5.5) == 211.0
```

Solution de la question 2

```python
def prix_ht(prix : float, taux : float) -> float:
    """Retourne le prix HT correspondant au prix TTC 'prix'
    avec un taux de TVA 'taux'.
    """
    return prix / (1 + taux / 100.0)
```

```python
# Jeu de tests
assert prix_ht(120, 20) == 100.0
assert prix_ht(100, 0) == 100.0
assert prix_ht(200, 100) == 100.0
assert prix_ht(0, 20) == 0.0
assert prix_ht(211, 5.5) == 200.0
```

Solution de l'exercice 1.3

Solution de la question 1

```python
def polynomiale(a : float, b : float, c : float, d : float, x : float) -> float:
    """Retourne la valeur de a*x^3 + b*x^2 + c*x + d
    """
    return (a*x*x*x + b*x*x + c*x + d)

# remarque : on peut aussi utiliser la fonction puissance :
# return (a*x**3 + b*x**2 + c*x + d)
# mais en considèrant que l'opérateur x**3 effectue lui-même x*x*x
# cela revient à faire le même nombre de multiplications.
```

```python
# Jeu de tests
assert polynomiale(1,1,1,1,2) == 15
assert polynomiale(1,1,1,1,3) == 40
assert polynomiale(2,0,0,0,1) == 2
assert polynomiale(0,3,0,0,1) == 3
assert polynomiale(0,0,4,0,1) == 4
assert polynomiale(1,2,3,4,0) == 4
assert polynomiale(2,3,4,5,1) == 14
```

Il faut 6 multiplications ici.

Une autre définition plus efficace, ne nécessitant que 3 multiplications, est la suivante :

```python
def polynomiale(a : float, b : float, c : float, d : float, x : float) -> float:
    """Retourne la valeur de a*x^3 + b*x^2 + c*x + d
    """
```

```python
    return (((((a*x + b) * x) + c) * x) + d)
```

```python
# Jeu de tests
assert polynomiale(1,1,1,1,2) == 15
assert polynomiale(1,1,1,1,3) == 40
assert polynomiale(2,0,0,0,1) == 2
assert polynomiale(0,3,0,0,1) == 3
assert polynomiale(0,0,4,0,1) == 4
assert polynomiale(1,2,3,4,0) == 4
assert polynomiale(2,3,4,5,1) == 14
```

Solution de la question 2

```python
def polynomiale_carre(a : float, b : float, c : float, x : float) -> float:
    """Retourne la valeur de a*x^4 + b*x^2 + c
    """
    return (a*x*x*x*x + b*x*x + c)

# ou
# return (a*x**4  + b*x**2 + c)
```

```python
# Jeu de tests
assert polynomiale_carre(1,1,1,2) == 21
assert polynomiale_carre(1,1,1,3) == 91
assert polynomiale_carre(2,0,0,1) == 2
assert polynomiale_carre(0,3,0,1) == 3
assert polynomiale_carre(2,3,4,0) == 4
assert polynomiale_carre(2,3,4,1) == 9
```

Il faut 6 multiplications ici aussi.

Une autre définition plus efficace (utilisant le schéma de *Hörner*), ne nécessitant plus que 4 multiplications, est la suivante :

```python
def polynomiale_carre(a : float, b : float, c : float, x : float) -> float:
    """Retourne la valeur de a*x^4 + b*x^2 + c
    """
    return (((a*x*x + b) * x*x) + c)
```

```python
# Jeu de tests
assert polynomiale_carre(1,1,1,2) == 21
assert polynomiale_carre(1,1,1,3) == 91
assert polynomiale_carre(2,0,0,1) == 2
assert polynomiale_carre(0,3,0,1) == 3
assert polynomiale_carre(2,3,4,0) == 4
assert polynomiale_carre(2,3,4,1) == 9
```

Solution de l'exercice 2.1

Solution de la question 1

```python
a = 3
```

Initialisation (première affectation) : la déclaration est :

```python
a : int
```

ou déclaration avec initialisation :

```python
a : int = 3
```

Table des variables :

Variable	`a`
Valeur	`3`

```python
a
```

Expression type `int` : valeur = `3`

```python
a = 0
```

Affectation (mise-à-jour).

Table des variables :

Variable	`a`
Valeur	`0`

```python
a
```

Expression type `int` : valeur = `0`

```python
a == 0
```

Expression type `bool` : valeur = `True`

```python
a == 3
```

Expression type `bool` : valeur = `False`

```python
b = 2
```

Initialisation (première affectation) : la déclaration est :

```python
b : int
```

Table des variables :

Variable	`a`	`b`
Valeur	`0`	`2`

```python
a == b
```

Expression type `bool` : valeur = `False`

```
b = a
```

Affectation.

Table des variables :

Variable	a	b
Valeur	0	0

```
b
```

Expression type `int` : valeur = `0`

```
a == b
```

Expression type `bool` : valeur = `True`

```
b = 4
```

Affectation.

Table des variables :

Variable	a	b
Valeur	0	4

```
a == b
```

Expression type `bool` : valeur = `False`

```
(a == 2) == (b == 0)
```

Expression type `bool` : valeur = `True`

```
c = (a == 0)
```

Initialisation (première affectation) : la déclaration est :

```
c : bool
```

Table des variables :

Variable	a	b	c
Valeur	0	4	True

```
c
```

Expression type `bool` : valeur = `True`

```
c == (b == 3)
```

Expression type `bool` : valeur = `False`

```
x
```

Expression : mais **erreur** car variable `x` non-initialisée.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
   ...
----> 1 x

NameError: name 'x' is not defined
```

Solution de la question 2

L'alternative n'est tout simplement pas correcte car le `if` attend une instruction booléenne et reçoit une instruction d'affectation. C'est la confusion classique entre l'instruction d'affection `=` et l'opérateur de test d'égalité `==`.

Il faut donc écrire en fait :

```
def sante(x : float) -> str:
    """Retourne "Bonne santé" si x vaut 37.5, retourne "Malade" sinon.
    """
    if x == 37.5:
        return "Bonne santé"
    else:
        return "Malade"
```

```
# Jeu de tests
assert sante(37.5) == "Bonne santé"
assert sante(40) == "Malade"
assert sante(36) == "Malade"
```

Solution de l'exercice 2.2

Solution de la question 1

Voici une première définition dans l'ordre croissant.

```
def mention(note : float) -> str:
    """Précondition : (note>=0) and (note<=20)
    Retourne la mention correspondant à la note spécifiée.
    """
    if note < 10:
        return 'Eliminé'
    elif note < 12:
        return 'Passable'
    elif note < 14:
        return 'AB'
    elif note < 16:
        return 'B'
    else:
        return 'TB'
```

```python
# Jeu de tests
assert mention(0) == 'Eliminé'
assert mention(8) == 'Eliminé'
assert mention(10) == 'Passable'
assert mention(12.5) == 'AB'
assert mention(15) == 'B'
assert mention(20) == 'TB'
```

Une autre version dans l'ordre décroissant :

```python
def mention(note : float) -> str:
    """ ... cf. ci-dessus ... """
    if note >= 16:
        return 'TB'
    elif note >= 14:
        return 'B'
    elif note >= 12:
        return 'AB'
    elif note >= 10:
        return 'Passable'
    else:
        return 'Eliminé'
```

Solution de la question 2

Dans la première réponse à la question 1, on fait 1 test lorsque $note < 10$, 2 tests lorsque $10 \leq note < 12$, 3 tests lorsque $12 \leq note < 14$ et 4 tests lorsque $14 \leq note \leq 20$. C'est statistiquement une version efficace si la majorité des notes sont mauvaises.

Dans la deuxième réponse à la question 1, on fait 4 tests lorsque $note < 12$, 3 tests lorsque $12 \leq note < 14$, 2 tests lorsque $14 \leq note < 16$ et 1 test lorsque $16 \leq note \leq 20$. C'est statistiquement une version efficace si la majorité des notes est bonne.

On va donner une version qui permet d'effectuer moins de tests si on suppose que la majorité des notes est entre 10 et 12 et qui fait moins de tests dans le pire des cas.

```python
def mention(note : float) -> str:
    """ ... cf-ci-dessus ..."""
    if note < 12:
        if note < 10:
            return 'Eliminé'
        else:
            return 'Passable'
    elif note < 14:
        return 'AB'
    elif note < 16:
        return 'B'
    else:
        return 'TB'
```

Avec cette version, l'évaluation ne nécessite que 2 tests lorsque $0 \leq note < 14$ et 3 tests lorsque $14 \leq note \leq 20$.

Solution de l'exercice 2.3

Solution de la question 1

```
# Jeu de tests
assert f(1, 2, 3) == 'cas 1'
assert f(1, 3, 2) == 'cas 2'
assert f(2, 1, 3) == 'cas 3'
assert f(3, 1, 2) == 'cas 4'
assert f(2, 3, 1) == 'cas 5'
assert f(3, 2, 1) == 'cas 6'
```

Solution de la question 2

```
def f(n1 : float, n2 : float, n3 : float) -> str:
    """ ... cf ci-dessus ...
    """
    if n1 < n2:
        if n2 < n3:
            return 'cas 1'
        elif n1 < n3:
            return 'cas 2'
        else:
            return 'cas 5'
    elif n1 < n3:
        return 'cas 3'
    elif n2 < n3:
        return 'cas 4'
    else:
        return 'cas 6'
```

Solution de l'exercice 2.4

Solution de la question 1

```
def egal_eps(x1 : float, x2 : float, epsilon : float) -> bool:
    """Précondition : epsilon > 0
    renvoie True quand x1 et x2 sont égaux à epsilon près.
    """
    return epsilon > abs (x1 - x2)
```

Discussion : Voici comment on peut *dénombrer* un ensemble de cas à tester en tenant compte des trois critères (signes des arguments, rapport des arguments, valeur attendue) :

- il y a 2 valeurs attendues (`True` et `False`) pour chacune d'elles,
- il y a 4 possibilités de signes:
  - 2 possibilités du même signe et alors, il y a pour chacune d'elle 2 possibilités de rapport entre les arguments: le premier est plus petit, le deuxième est plus petit
  - 2 possibilités de signes différents et alors le rapport est fixé.

Ce qui donne $2 * (2 * 2 + 2) = 12$ cas à envisager.

Par exemple:

```
# Jeu de tests
assert egal_eps(4,5,1.1) == True
assert egal_eps(5,4,1.1) == True
assert egal_eps(-4,-5,1.1) == True
assert egal_eps(-5,-4,1.1) == True
assert egal_eps(3,5,1.1) == False
assert egal_eps(5,3,1.1) == False
assert egal_eps(-3,-5,1.1) == False
assert egal_eps(-5,-3,1.1)== False
assert egal_eps(-1,0,1.1) == True
assert egal_eps(0,-1,1.1) == True
assert egal_eps(-1,1,1.1) == False
assert egal_eps(1,-1,1.1) == False
```

Solution de la question 2

Il n'y a qu'une seule façon d'obtenir $\frac{3}{3}$ (c'est-à-dire 1) : les trois valeurs sont égales à *epsilon* près deux à deux.

Il y a trois façons d'obtenir $\frac{2}{3}$: soit `v1` est égale à *epsilon* près à `v2` et à `v3` ; soit `v2` est égale à *epsilon* près à `v1` et à `v3` ; soit `v3` est égale à *epsilon* près à `v1` et à `v2`.

Il y a quatre façons d'obtenir 0 : soit les valeurs sont toutes différentes, soit l'une des égalités à *epsilon* près est vérifiée mais aucune des deux autres n'est possible et il y a trois possibilités pour cela : soit `v1` est égale à *epsilon* près à `v2` mais ni `v1` à `v3`, ni `v2` à `v3` ; soit `v2` est égale à *epsilon* près à `v3` mais ni `v1` à `v3`, ni `v1` à `v2` ; soit `v2` est égale à *epsilon* près à `v3` mais ni `v1` à `v3`, ni `v1` à `v2`.

```
def fiabilite(v1 : float, v2 : float, v3 : float
              , epsilon : float) -> float:
    """Précondition: epsilon > 0
    Retourne le taux de fiabilité des trois mesures v1, v2 et v3
    à epsilon près.
    """
    if egal_eps(v1, v2, epsilon) and egal_eps(v2 ,v3 , epsilon) \
        and egal_eps(v1, v3, epsilon):
        return 1
    elif ((egal_eps(v1, v2, epsilon) and egal_eps(v2,v3,epsilon))
        or (egal_eps(v1, v2, epsilon) and egal_eps(v1, v3, epsilon))
        or (egal_eps(v1, v3, epsilon) and egal_eps(v2, v3, epsilon))):
        return 2/3
    else:
        return 0
```

```
# Jeu de tests
assert fiabilite(3,3,3,1.1) == 1
assert fiabilite(3,2,1,1.1) == 2/3
assert fiabilite(1,2,3,1.1) == 2/3
assert fiabilite(1,3,2,1.1) == 2/3
```

```python
assert fiabilite(1,3,5,1.1) == 0
assert fiabilite(1,2,5,1.1) == 0
assert fiabilite(1,5,2,1.1) == 0
assert fiabilite(5,1,2,1.1) == 0
```

Deuxième Solution : La solution précédente, si elle calcule bien ce que l'on veut, effectue possiblement plusieurs fois un même test, ce que l'on peut éviter en imbriquant les conditionnelles.

```python
def fiabilite(v1 : float, v2 : float, v3 : float
              , epsilon : float) -> float:
    """ ... cf. ci-dessus ...
    """
    if egal_eps(v1, v2, epsilon):
        if egal_eps(v1, v3, epsilon):
            if egal_eps(v2, v3, epsilon):
                return 1
            else:
                return 2/3
        else:
            if egal_eps(v2, v3, epsilon):
                return 2/3
            else:
                return 0
    elif egal_eps(v2, v3, epsilon):
        if egal_eps(v1, v3, epsilon):
            return 2/3
        else:
            return 0
    else:
        return 0
```

Troisième Solution : Bien sûr, on peut aussi précalculer nos égalités *à epsilon près*. Cette solution est peut-être plus simple mais elle effectue des calculs parfois inutiles.

```python
def fiabilite(v1 : float, v2 : float, v3 : float
              , epsilon : float) -> float:
    """ ... cf. ci-dessus ...
    """
    egal_v1_v2 : bool = egal_eps(v1, v2, epsilon)

    egal_v2_v3 : bool = egal_eps(v2, v3, epsilon)

    egal_v1_v3 : bool = egal_eps(v1, v3, epsilon)

    if egal_v1_v2 and egal_v2_v3 and egal_v1_v3:
        return 1
    elif ((egal_v1_v2 and egal_v2_v3)
        or (egal_v1_v2 and egal_v1_v3)
        or (egal_v1_v3 and egal_v2_v3)):
        return 2/3
```

```
    else:
        return 0
```

Solution de l'exercice 3.1

Solution de la question 1

```
def somme_impairs_inf(n : int) -> int:
    """Précondition: n >= 0
    Renvoie la somme de tous les entiers naturels impairs
    inférieurs ou égaux à n.
    """
    # somme calculée
    s : int = 0

    # impair courant  (1 est le premier impair)
    i : int = 1

    while i <= n:
        s =  s + i
        i = i + 2

    return s
```

```
# Jeu de test
assert somme_impairs_inf(0) == 0
assert somme_impairs_inf(1) == 1
assert somme_impairs_inf(2) == 1
assert somme_impairs_inf(3) == 4
assert somme_impairs_inf(4) == 4
assert somme_impairs_inf(5) == 9
assert somme_impairs_inf(8) == 16
```

Solution de la question 2

```
def somme_premiers_impairs(n : int) -> int:
    """Précondition : n > 0
    Renvoie la somme des n premiers entiers impairs.
    """
    # somme calculée
    s : int = 0

    # compteur
    i : int = 1

    # impair courant (1 est le premier impair)
    imp : int = 1

    while i <= n:
```

```
        s = s + imp
        imp = imp + 2
        i = i + 1

    return s
```

```
# Jeu de tests
assert somme_premiers_impairs(1) == 1 ** 2
assert somme_premiers_impairs(2) == 2 ** 2
assert somme_premiers_impairs(3) == 3 ** 2
assert somme_premiers_impairs(4) == 4 ** 2
assert somme_premiers_impairs(5) == 5 ** 2
assert somme_premiers_impairs(8) == 8 ** 2
assert somme_premiers_impairs(9) == 9 ** 2
```

Solution de la question 3

Tour de boucle	variable `s`	variable `imp`	variable `i`
entrée	**0**	**1**	**1**
1er	1	3	2
2e	4	5	3
3e	9	7	4
4e	16	9	5
5e (sortie)	**25**	**11**	**6**

Remarque : dans le corps de la boucle la première variable affectée est `s`, puis `imp` et enfin `i`. L'ordre des colonnes dans la table de simulation suit donc cet ordre, ce qui permet une lecture de gauche-à-droite et de haut-en-bas, comme expliqué précédemment.

Solution de l'exercice 3.2

Solution de la question 1

```
def f(x : int, y : int) -> int:
    """ int * int -> int

    ??? mystère !"""

    z : int = 0

    w : int = x

    while w <= y:
        z = z + w * w
        w = w + 1

    return z
```

Solution de la question 2

Tour de boucle	variable `z`	variable `w`
entrée	**0**	**3**
1e	9	4
2e	25	5
3e	50	6
4e (sortie)	**86**	**7**

Remarque : la colonne de la variable `z` est à gauche de la colonne de la variable `w` car *dans le corps de la boucle* l'affectation à `z` précède l'affecation à `w`.

```
>>> f(3,6)
86
```

Solution de la question 3

La boucle incrémente la variable `w` à chaque passage. La condition devient donc fausse quand $w > y$, c'est-à-dire lorsque w vaut $y + 1$.

Solution de la question 4

1. Avant le premier passage dans la boucle, `w` vaut 5 (valeur de `x`), `z` vaut 0 et `y` vaut 3.
2. La condition de la boucle `while` $w \leq y$ est donc immédiatement fausse. On ne fait aucun passage.
3. La valeur de retour est donc 0, valeur initiale de `z`.

Donc il semble pertinent d'ajouter l'hypothèse d'appel suivante :

```
def f(x : int, y : int) -> int:
    """Précondition:  x <= y
    ...
    """
```

Solution de la question 5

```
def somme_carres(m : int, n : int) -> int:
    """Précondition: m <= n
    Retourne la somme des carrés des entiers  dans l'invervalle [m;n].
    """

    # la somme des carrés
    s : int = 0

    # entier courant dans l'intervalle
    i : int = m

    while i <= n:
        s = s + i * i
        i = i + 1

    return s
```

```python
# Jeu de tests
assert somme_carres(1, 5) == 55
assert somme_carres(2, 5) == 54
assert somme_carres(3, 5) == 50
assert somme_carres(4, 5) == 41
assert somme_carres(5, 5) == 25
assert somme_carres(3, 6) == 86
assert somme_carres(-4, 0) == 30
```

Solution de l'exercice 3.3

Solution de la question 1

```python
def divise(n : int, p : int) -> bool:
    """Précondition : n > 0 et p >= 0
    Renvoie True si et seulement si n divise p.
    """
    return p % n == 0
```

```python
# Jeu de tests
assert divise(1, 4) == True
assert divise(2, 4) == True
assert divise(3, 4) == False
assert divise(4, 4) == True
assert divise(4, 2) == False
assert divise(17, 123) == False
assert divise(17, 357) == True
assert divise(21, 357) == True
```

Solution de la question 2

Première définition sans sortie anticipée :

```python
def est_premier(n : int) -> bool:
    """Précondition: n >= 0
    renvoie True si et seulement si n est premier.
    """
    if n < 2:
        return False
    else:
        # pas de diviseur trouvé ?
        b : bool = True

        # prochain diviseur potentiel
        i : int = 2

        while b and (i < n):
            if divise(i, n):
                b = False
```

```python
            else:
                i = i + 1

        return b
```

```python
# Jeu de tests
assert est_premier(0) == False
assert est_premier(1) == False
assert est_premier(2) == True
assert est_premier(17) == True
assert est_premier(357) == False
```

Deuxième définition avec sortie anticipée :

```python
def est_premier(n : int) -> bool:
    """ ... cf. ci-dessus ...
    """
    if n < 2:
        return False
    else:
        # prochain diviseur potentiel
        i : int = 2
        while i < n:
            if divise(i, n):
                return False
            else:
                i = i + 1

        return True
```

Solution de l'exercice 4.1

Solution de la question 1

```python
# Jeu de tests
assert factorielle(0) == 1
assert factorielle(1) == 1
assert factorielle(2) == 2
assert factorielle(3) == 6
assert factorielle(4) == 24
assert factorielle(5) == 120
assert factorielle(6) == 720
```

Solution de la question 2

Tour de boucle	variable `f`	variable `k`
entrée	**1**	**1**
1er	1	2
2e	2	3

Tour de boucle	variable `f`	variable `k`
3e	6	4
4e	24	5
5e (sortie)	**120**	**6**

Solution de la question 3 : à propos de la correction

Rappel : l'invariant doit être une expression booléenne vraie en entrée de boucle ainsi qu'après chaque tour de boucle.

Après quelques minutes (courtes) de réflexion, si aucun étudiant ne propose ...

candidat **invariant de boucle** : $f = (k-1)!$ (ou $f = \prod_{i=1}^{k-1} i$).

Remarque : l'invariant est une expression mathématique, pas une expression python.

Pour vérifier l'invariant sur la simulation, on ajoute simplement une colonne dédiée :

Tour de boucle	variable `f`	variable `k`	**invariant** $f = (k-1)!$
entrée	**1**	**1**	$1 = (1-1)!$ (vrai par convention)
1er	1	2	$1 = (2-1)!$ (vrai)
2e	2	3	$2 = (3-1)!$ (vrai)
3e	6	4	$6 = (4-1)!$ (vrai)
4e	24	5	$24 = (5-1)!$ (vrai)
5e (sortie)	**120**	**6**	$120 = (6-1)!$ (vrai)

On a donc vérifié notre candidat invariant pour `n=5`. On peut montrer que cet invariant de boucle est vérifié quel que soit `n` satisfaisant l'hypothèse `n > 0` mais cela dépasse le cadre du cours.

Si on en fait l'hypothèse alors en sortie de boucle on a $k = n + 1$ donc l'invariant devient $f = (n + 1 - 1)! = n!$. Comme la fonction retourne f elle calcule bien $n!$.

En conclusion notre fonction est correcte.

Solution de la question 4 : à propos de la terminaison

Rappel : le variant doit être une expression arithmétique sur les entiers naturels

- qui décroit strictement après chaque tour de boucle
- qui vaut 0 lorsque la condition de boucle devient fausse (donc en sortie de boucle)

Après quelques minutes (courtes) de réflexion, si aucun étudiant ne propose ...

Candidat **variant de boucle** : $n + 1 - k$

Tour de boucle	variable `f`	variable `k`	**variant** $n + 1 - k$
entrée	**1**	**1**	**5**
1er	1	2	4
2e	2	3	3
3e	6	4	2

Tour de boucle	variable `f`	variable `k`	**variant** $n+1-k$
4e	24	5	1
5e (sortie)	**120**	**6**	0

Donc le variant de boucle est vérifié pour `n=5`.

Informellement, on sait de plus que :

— $n \geq 0$ et $k = 0$ en entrée de boucle donc $n+1-k$ est positif.
— le variant $n+1-k$ décroit strictement après chaque tour de boucle.
— et la plus petit valeur de k pour laquelle $k > n$ (la condition de boucle est fausse) est $k = n+1$ donc exactement lorsque le variant $n+1-k = 0$.

On peut donc en conclure que la boucle termine pour n'importe quelle valeur de `n` satisfaisant l'hypothèse `n > 0`.

Solution de l'exercice 4.2

Solution de la question 1

```python
def f(n : int, m : int) -> int:
    a : int = n
    b : int = 0
    c : int = 0

    while a > 0:
        while a > 0:
            a = a - 1
            b = b + 1

        a = b - 1
        b = 0
        c = c + m

    return c
```

Solution de la question 2

```python
>>> f(2, 1)
2
```

```python
>>> f(3, 10)
30
```

```python
>>> f(0, 8)
0
```

```python
>>> f(5, 0)
0
```

```
>>> f(6, 7)
42
```

La lecture de la simulation n'est pas difficile :

- l'entrée de la boucle extérieure est comme pour les boucles simples.
- il y a une entrée de boucle intérieure :
  - après l'entrée de la boucle extérieure (ici, il n'y a pas d'affectation entre les deux `while` mais ce n'est bien sûr pas le cas en général)
  - après chaque fin de tour la boucle extérieure
- le tiret de la boucle intérieur correspond au code qui est :
- soit entre les deux `while` (ici vide mais il faut compter l'entrée du `while extérieur`)
- soit entre la fin du *dernier* tour de la boucle intérieure et la prochaine entrée.

Solution de la question 3

D'après la simulation précédente, la fonction `f` ajoute la valeur de `m` à `c` autant de fois que l'on réinitialise `a`. Comme on réinitialise `a` en le décrémentant, on conjecture donc que la fonction renvoie **`n` fois `m`** et réalise donc la multiplication de ses entrées.

```
>>> f(3, -4)
-12
```

```
>>> f(-3, 4)
0
```

On trouve donc raisonnable de restreindre la fonction aux entiers naturels.

```python
def mult(n : int, m : int) -> int:
    """Précondition : (n >= 0) and (m >= 0)
    Renvoie la multiplication de n par m.
    """
    a : int = n
    b : int = 0
    c : int = 0

    while a > 0:
        while a > 0:
            a = a - 1
            b = b + 1

        a = b - 1
        b = 0
        c = c + m

    return c
```

```python
# Jeu de tests
assert mult(3, 4) == 12
assert mult(2, 1) == 2
assert mult(3, 10) == 30
assert mult(8, 0) == 0
assert mult(0, 5) == 0
```

```python
assert mult(6, 7) == 42
assert mult(9, 99) == 891
```

Solution de l'exercice 4.3

Solution de la question 1 - Simulation

Le couple `(54,15)` vérifie bien les hypothèses de la fonction: $54 \geq 15 > 0$.

La variable temporaire `temp` servant uniquement à stocker une valeur pour `r`, on ne reporte pas sa valeur dans le tableau.

Tour de boucle	variable `q`	variable `r`
entrée	**54**	**15**
1er	15	9
2e	9	6
3e	6	3
4e (sortie)	**3**	**0**

```python
>>> pgcd(54, 15)
3
```

Solution de la question 2 - Correction

Vérifions l'invariant sur la simulation:

Tour	variable `q`	variable `r`	$\mathbf{div}(q, r)$	$q \geq r \geq 0$	Invariant
entrée	**54**	**15**	$\mathbf{div}(54, 15) = \{1, 3\}$	$54 \geq 15 \geq 0$	Vrai
1er	15	9	$\mathbf{div}(15, 9) = \{1, 3\}$	$15 \geq 9 \geq 0$	Vrai
2e	9	6	$\mathbf{div}(9, 6) = \{1, 3\}$	$9 \geq 6 \geq 0$	Vrai
3e	6	3	$\mathbf{div}(6, 3) = \{1, 3\}$	$6 \geq 3 \geq 0$	Vrai
4e (sortie)	**3**	**0**	$\mathbf{div}(3, 0) = \{1, 3\}$	$3 \geq 0 \geq 0$	Vrai

On a donc vérifié notre candidat invariant pour le couple d'entrée `(54,15)`.

On peut montrer que cet invariant de boucle est vérifié quels que soient `a,b` satisfaisant les hypothèses `n > 0`.

Supposons `a,b` tels que $a \geq b > 0$.

L'invariant est vrai au début de la fonction. En effet, comme $q = a$ et $b = r$, on a trivialement $\mathbf{div}(q, r) = \mathbf{div}(a, b)$ et l'hypothèse de départ implique $(q \geq r \geq 0)$.

Supposons que l'invariant est vrai au début d'une boucle. On a $\mathbf{div}(a, b) = \mathbf{div}(q, r) \wedge (q \geq r \geq 0)$. La condition de boucle nous indique en plus que $r \neq 0$.

Appelons `qq` et `rr` les valeurs respectives de `q` et `r` en fin de boucle. On a `qq = r` et `rr = q % r`.

Soit d un diviseur de q et r. Trivialement, d divise qq. La définition du modulo nous permet d'écrire $q = r * s + rr$ avec $0 \leq rr < r$. Comme d divise q, d divise la somme $r * s + rr$, et comme d divise $r * s$ par hypothèse, alors d divise rr.

Soit d un diviseur de qq et rr. Trivialement, d divise r. La définition du modulo nous permet d'écrire $q = r * s + rr$ avec $0 \leq rr < r$. Comme d divise rr et d divise $r * s$ (car il divise r), alors d divise q.

On vient de montrer que les diviseurs communs à q et r sont les mêmes que ceux communs à qq et rr. On a supposé que $\mathbf{div}(a, b) = \mathbf{div}(q, r)$, on a montré que $\mathbf{div}(qq, rr) = \mathbf{div}(q, r)$, on en conclut que $\mathbf{div}(a, b) = \mathbf{div}(qq, rr)$.

On a supposé que $q \geq r \geq 0$, et on a, par définition du modulo, $q = r * s + rr$ avec $0 \leq rr < r$. On conclut que $qq = r \geq rr \geq 0$.

Ainsi, on vient de montrer que si on suppose l'invariant vrai au début d'une boucle, il est vrai en fin de boucle.

Si l'invariant est vrai en sortie de boucle, on a (entre autres) $\mathbf{div}(q, r) = \mathbf{div}(a, b)$. La condition de sortie de boucle nous indique que $r = 0$, ainsi $\mathbf{div}(q, r)$ est égal à l'ensemble des diviseurs de q. Et l'invariant devient $\mathbf{div}(q) = \mathbf{div}(a, b)$, c'est-à-dire "l'ensemble des diviseurs communs à a et b et l'ensemble des diviseurs de q", ce qui équivaut à "q est le plus grand diviseur commun à a et b".

En conclusion notre fonction est correcte.

Solution de la question 3 - Terminaison

Le **variant de boucle** de plus évident est : r

On a déjà la valeur de la variable `r` dans la simulation effectuée précédemment :

Tour de boucle	variable `r`
entrée	**15**
1er	9
2e	6
3e	3
4e (sortie)	**0**

Donc le variant de boucle est vérifié pour le couple d'entrées `54,15`.

Montrons que `pgcd(a,b)` termine. On utilise pour cela la deuxième partie de l'invariant de la fonction précédente: on sait qu'à tout moment $q \geq r \geq 0$.

Comparons la valeur du variant au début et à la fin d'une boucle. Appelons `rr` la valeur de `r` en fin de boucle. On sait que `rr = q % r`. On a, par définition du modulo, $q = r * s + rr$ avec $q > rr \geq 0$. On sait par l'invariant que $q \geq r$, on en déduit que s ne peut valoir 0 (sinon on a $rr = q > rr$). L'invariant nous dit aussi que r et q sont positifs, et on en déduit que $rr < r$.

Plus simplement, on peut dire que dans `rr = q % r` si `r` est non nul, alors `rr < r` par définition du modulo.

On vient de montrer que r décroit strictement à chaque tour de boucle. Comme la condition de sortie de boucle est $r = 0$, toute exécution finit par sortir de la boucle, et `pgcd` termine.

Solution de l'exercice 5.1

Solution de la question 1

```
def somme_carres(n : int) -> int:
    """Précondition : n >= 0
    Retourne la somme des carrés des entiers inférieurs ou
    égaux à n.
    """
    # Somme à calculer
    s : int = 0

    i : int # Entier courant
    for i in range(1, n + 1):
        s = s + i * i

    return s
```

```
# Jeu de tests
assert somme_carres(0) == 0
assert somme_carres(1) == 1
assert somme_carres(2) == 5
assert somme_carres(3) == 14
assert somme_carres(4) == 30
assert somme_carres(5) == 55
```

Solution de la question 2

Cette fonction calcule le produit des cubes des entiers k dans l'intervalle $[m, n[$.

$$f(m, n) = \prod_{k=m}^{n-1} k^3$$

```
def produit_cubes(m : int, n : int) -> int:
    """Précondition : (0 <= m) and (m <= n)
    Retourne le produit des cubes des entiers dans l'intervalle [m,n[.
    """
    # Produit à calculer
    p : int = 1

    k : int # Entier courant
    for k in range(m, n):
        p = p * k * k * k

    return p
```

```
# Jeu de tests
assert produit_cubes(1, 4) == 1 * 8 * 27
assert produit_cubes(2, 4) == 216
```

```python
assert produit_cubes(4, 8) == 592704000
```

Simulation de `produit_cubes(4, 8)` :

Tour de boucle	variable `k`	variable `p`
entrée	-	**1**
1er	4	64
2e	5	8000
3e	6	1728000
4e	7	592704000
sortie	-	**592704000**

Solution de l'exercice 5.2

Solution de la question 1

```python
def f(a : str) -> int:
    b : int = 0
    c : str
    for c in a:
        if c >= '0' and c <= '9':
            b = b + 1

    return b
```

Solution de la question 2

Tour de boucle	variable `c`	variable `b`
entrée	-	0
1e	'1'	1
2e	'0'	2
3e	' '	2
4e	'a'	2
5e	'o'	2
6e	'û'	2
7e	't'	2
sortie	-	2

```python
>>> f('bonjour')
0
```

```python
>>> f('un : 1')
1
```

```python
>>> f('606060')
6
```

Solution de la question 3

D'après la simulation précédente, la fonction `f` ajoute 1 à `b` à chaque fois que le caractère `c` est un chiffre. On conjecture donc que la fonction compte le nombre de chiffres dans la chaîne donnée en entrée.

```
def nb_chiffres(s : str) -> int:
    """Renvoie le nombre de chiffres de s.
    """
    nb : int = 0

    c : str
    for c in s:
        if c >= '0' and c <= '9':
            nb = nb + 1

    return nb
```

```
# Jeu de tests
assert nb_chiffres('bonjour') == 0
assert nb_chiffres('12345') == 5
assert nb_chiffres('0') == 1
assert nb_chiffres('') == 0
assert nb_chiffres('a1b2c3ed4') == 4
```

Solution de l'exercice 5.3

Solution de la question 1

```
def est_palindrome(s : str) -> bool:
    """Retourne True si et seulement si s est un palindrome.
    """
    i : int # Indice courant
    for i in range(0, len(s)):
        if s[i] != s[len(s)-i-1]:
            return False  # différence trouvée

    # La boucle s'est terminée sans qu'on trouve de différence
    # entre la lecture dans un sens et dans l'autre
    return True
```

```
# Jeu de tests
assert est_palindrome('') == True
assert est_palindrome('je ne suis pas un palindrome') == False
assert est_palindrome('aaaa') == True
assert est_palindrome('aba') == True
assert est_palindrome('amanaplanacanalpanama') == True
```

Variante plus efficace (contrôle que la moitié de gauche correspond à la moité de droite)

```python
def est_palindrome(s : str) -> bool:
    """ ... cf. ci-dessus ...
    """
    i : int
    for i in range(0, len(s)//2):
        if s[i] != s[len(s)-i-1]:
            return False

    return True
```

Une variante de cette dernière fonction qui exploite l'utilisation des entiers négatifs pour l'accès aux éléments d'une chaîne de caractères (vu en cours) :

```python
def est_palindrome(s : str) -> bool:
    """ ... cf. ci-dessus ...
    """
    i : int
    for i in range(0,len(s)//2):
        if s[i] != s[-i-1]:
            return False

    return True
```

Solution de la question 2

Remarque : dans le jeu de tests on exploitera le fait que la chaîne miroir est un palindrome.

```python
def miroir(s : str) -> str:
    """Retourne le palindrome miroir de la chaîne s.
    """
    # Chaîne inversée
    r : str = ''

    ch : str  # Caractère courant
    for ch in s:
        r = ch + r

    return s + r
```

```python
# Jeu de tests
assert miroir('abc') == 'abccba'
assert est_palindrome(miroir('abc'))
assert est_palindrome(miroir('amanaplanacanal'))
assert est_palindrome(miroir('do-re-mi-fa-sol'))
```

Variante exploitant les indices négatifs dans les chaînes de caractères :

```python
def miroir(s : str) -> str:
    """ ... cf. ci-dessus ...
    """
    # Chaîne inversée
    r : str = ''
```

```python
    i : int  # Position du caractère courant
    for i in range(1,len(s)+1):
        r = r + s[-i]

    return s + r
```

Solution de l'exercice 5.4

Solution de la question 1

```python
def suppression_debut(c : str, s : str) -> str:
    """Précondition : len(c) == 1
    Retourne la chaîne s sans la première occurrence du caractère c.
    """
    # Indicateur si la première occurrence
    # de c a été vue ou non
    premiere_trouvee : bool = False

    # Résultat
    res : str = ''

    d : str
    for d in s:
        if d != c:
            res = res + d
        elif  not premiere_trouvee:
            premiere_trouvee = True
        else:
            res = res + d

    return res
```

```python
# Jeu de tests
assert suppression_debut('a', '') == ''
assert suppression_debut('a', 'aaaaa') == 'aaaa'
assert suppression_debut('p', 'le papa noel') == 'le apa noel'
assert suppression_debut('a', 'bbbbb') == 'bbbbb'
```

Solution de la question 2

```python
def suppression_derniere(c : str, s : str) -> str:
    """Précondition : len(c) == 1
    Retourne la chaîne s sans la dernière occurrence du caractère c.
    """

    # Indicateur si la dernière occurrence
    # de c a été vue ou non
    derniere_trouvee : bool = False
```

```python
    # Résultat
    res : str = ''

    # Indice du caractère courant, en partant de la fin
    i : int = len(s) - 1
    while i >= 0:
        if s[i] != c:
            res = s[i] + res
        elif  not derniere_trouvee:
            derniere_trouvee = True
        else:
            res = s[i] + res

        i = i - 1

    return res
```

```python
# Jeu de tests
assert suppression_derniere('a','') == ''
assert suppression_derniere('a', 'aaaaa') == 'aaaa'
assert suppression_derniere('p', 'le papa noel') == 'le paa noel'
assert suppression_derniere('a', 'bbbbb') == 'bbbbb'
```

Solution de l'exercice 6.1

Solution de la question 1

```python
def repetition(x : T, k : int) -> List[T]:
    """Précondition : k >= 0
    Retourne la liste composée de k occurrences de x.
    """
    # liste résultat
    lr : List[T] = []

    i : int
    for i in range(0, k):
        lr.append(x)

    return lr
```

```python
# Jeu de tests
assert repetition(0, 4) == [0, 0, 0, 0]
assert repetition(4, 0) == []
assert repetition('pom', 5) == ['pom', 'pom', 'pom', 'pom', 'pom']
```

Solution de la question 2

```python
def repetition_bloc(l : List[T], k : int) -> List[T]:
    """Précondition : k >= 0
```

```
    Retourne la liste composée de k répétitions de la liste l.
    """
    # Liste résultat
    lr : List[T] = []

    i : int
    for i in range(k):
        lr = lr + l

    return lr
```

```
# Jeu de tests
assert repetition_bloc([1, 2], 4) == [1, 2, 1, 2, 1, 2, 1, 2]
assert repetition_bloc([4, 5, 2, 3], 0) == []
assert repetition_bloc(['pim', 'pam', 'poum'], 3) \
        ==  ['pim', 'pam', 'poum', \
             'pim', 'pam', 'poum', \
             'pim', 'pam', 'poum']
```

Remarque : en Python l'opérateur `*` permet de construire des listes de répétitions. Par exemple :

```
>>> ["thon"] * 4

['thon', 'thon', 'thon', 'thon']
```

```
>>> 8 * [3]

[3, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> 0 * [1, 2, 3, 4, 5]
[]
```

Solution de l'exercice 6.2

Solution de la question 1

```
def max_liste(l : List[float]) -> float:
    """Précondition :  len(l) > 0
    Retourne le plus grand élément de la liste L.
    """
    # Maxium partiel
    mx : float = l[0] # cf. précondition

    e : float
    for e in l[1:]:
        if e > mx:
            mx = e
```

```python
    return mx
```

```python
# Jeu de tests
assert max_liste([3, 7, 9, 5.4, 8.9, 9, 8.999, 5]) == 9
assert max_liste([-2, -1, -5, -3, -1, -4, -1]) == -1
```

Solution de la question 2

```python
def nb_occurrences(l : List[T], x : T) -> int:
    """Retourne le nombre d'occurrences de x dans L.
    """
    # Résultat
    res : int = 0 # (la valeur calculée)

    e : T
    for e in l:
        if e == x:
            res = res + 1

    return res
```

```python
# Jeu de tests
assert nb_occurrences([3, 7, 9, 5.4, 8.9, 9, 8.999, 5], 9) == 2
assert nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "chat") \
         == 3
assert nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "ou") \
         == 0
```

Solution de la question 3

```python
# la solution "triviale" est d'utiliser les 2 fonctions précédentes:
def nb_max(l : List[float]) -> int:
    """Précondition : len(l) > 0
    Retourne le nombre d'occurrences du maximum de l dans l.
    """
    return nb_occurrences(l, max_liste(l))

# Pouvez-vous trouver une version plus efficace ne nécessitant qu'un seul parcours ?
# (réponse au chapitre 7)
```

```python
# Jeu de tests
assert nb_max([3, 7, 9, 5.4, 8.9, 9, 8.999]) == 2
assert nb_max([-2, -1, -5, -3, -1, -4, -1]) == 3
```

Solution de l'exercice 6.3

Solution de la question 1

```python
def liste_diviseurs(a : int) -> List[int]:
    """Précondition : a > 0
    Retourne la liste des diviseurs de a."""
```

```python
    # liste résultat
    lr : List[int] = []

    i : int
    for i in range(1, a + 1):
        if a % i == 0:
            lr.append(i)
    return lr
```

```python
# Jeu de tests
assert liste_diviseurs(2) == [1, 2]
assert liste_diviseurs(12) == [1, 2, 3, 4, 6, 12]
assert liste_diviseurs(25) == [1, 5, 25]
```

Solution de la question 2

```python
def liste_diviseurs_impairs(a : int) -> List[int]:
    """Précondition : a > 0
    Retourne la liste des diviseurs impairs de a.
    """
    # Liste résultat
    lr : List[int] = []

    # Candidat diviseur impair
    i : int = 1 # 1 est le plus petit candidat possible.

    while i < a + 1:
        if a % i == 0:
            lr.append(i)

        i = i + 2

    return lr
```

```python
# Jeu de tests
assert liste_diviseurs_impairs(2) == [1]
assert liste_diviseurs_impairs(12) == [1, 3]
assert liste_diviseurs_impairs(30) == [1, 3, 5, 15]
assert liste_diviseurs_impairs(15) == [1, 3, 5, 15]
```

Solution de l'exercice 6.4

Solution de la question 1

```python
def f(l : List[float]) -> bool:
    # etc ...
```

Solution de la question 2

Simulation de boucle pour `f([3, 5, 7, 10])`:

tour de boucle	variable `i`	`l[i]`	`l[i+1]`
entrée	**0**	**3**	**5**
1er tour	1	5	7
2e	2	7	10
sortie	-	-	-

Sortie « normale » de boucle. Instruction suivante : `return True`. La valeur renvoyée est `True`.

Simulation de boucle pour `f([3, 15, 7, 10])`:

tour de boucle	variable `i`	`l[i]`	`l[i+1]`
entrée	**0**	**3**	**15**
1er tour	1	15	7
2e	2	7	10
sortie anticipée	-	-	-

Sortie « anticipée » de boucle. La valeur renvoyée est `False`.

Solution de la question 3

```python
def est_croissante(l : List[float]) -> bool:
    """Retourne True si l est rangée en ordre strictement croissant
    et False sinon.
    """
    if (len(l) == 0) or (len(l) == 1):
        return True
    else:
        i : int
        for i in range(len(l) - 1):
            if l[i] >= l[i + 1]:
                return False
    return True
```

```python
# Jeu de tests
assert est_croissante([1, 3, 4, 7, 8, 11, 13]) == True
assert est_croissante([1, 3, 4, 7, 8, 11, 9]) == False
assert est_croissante([1, 3, 4, 2, 5, 6]) == False
assert est_croissante([1, 3, 3, 4, 5, 6]) == False
assert est_croissante([]) == True
assert est_croissante([5]) == True
```

Solution de la question 4

```python
def est_croissante(l : List[float]) -> bool:
    """ ... cf. ci-dessus ...
    """
    # Résultat
    b : bool = True
    # Indice courant
```

```python
    i : int = 0
    while (i < len(l) - 1) and b:
        b = l[i] < l[i + 1]
        i = i + 1
    return b
```

Solution de l'exercice 6.5

Solution de la question 1 : découpages simples

```python
def decoupage_simple(l : List[T], i : int, j : int) -> List[T]:
    """Précondition : (i >= 0) and (j >= 0)
    Retourne le découpage l[i:j].
    """
    # Liste résultat
    lr : List[T] = []

    k : int
    for k in range(i, j):
        if k < len(l):
            lr.append(l[k])

    return lr
```

Solution de la question 2 : découpage avec pas

```python
def decoupage_pas(l : List[T], i : int, j : int, p : int) -> List[T]:
    """Précondition : (i >= 0) and (j >= 0) and (p > 0)
    Retourne le découpage l[i:j:p].
    """
    lr : List[T] = []
    k : int
    k = i   # on commence en i
    while k < j:   # j n'est pas inclus
        if k < len(l):
            lr.append(l[k])

        k = k + p

    return lr
```

Solution de la question 3 : pas inverse

```python
def decoupage_pas_inv(l : List[T], i : int, j : int, p : int) -> List[T]:
    """Précondition : (i >= 0) and (j >= 0) and (p < 0)
    Retourne le découpage l[i:j:p].
    """
    lr : List[T] = []
    k : int = i  # on commence en i
    while k > j:   # j est inclus
```

```python
        if k < len(l):
            lr.append(l[k])

        k = k + p

    return lr
```

Solution de la question 4 : découpage généralisé

```python
def decoupage(l : List[T], i : int, j : int, p : int) -> List[T]:
    """Précondition : p != 0
    Retourne le découpage l[i:j:p].
    """
    # Normalisation de i
    ni : int = normalisation(i, len(l))

    # Normalisation de j
    nj : int = normalisation(j, len(l))

    if p > 0: # pas strictement positif
        return decoupage_pas(l, ni, nj, p)
    else: # pas strictement négatif (d'après l'hypothèse)
        return decoupage_pas_inv(l, ni, nj, p)
```

```python
# Jeu de tests
assert decoupage(lcomptine, 1, 3, 1) == lcomptine[1:3]
assert decoupage(lcomptine, 3, 4, 1) == lcomptine[3:4]
assert decoupage(lcomptine, 3, 3, 1) == lcomptine[3:3]
assert decoupage(lcomptine, 5, 3, 1) == lcomptine[5:3]
assert decoupage(lcomptine, 0, 7, 1) == lcomptine[0:7]
assert decoupage(lcomptine, -4, -1, 1) == lcomptine[-4:-1]
assert decoupage(lcomptine, -6, -2, 1) == lcomptine[-6:-2]
assert decoupage(lcomptine,1, 5, 2) == lcomptine[1:5:2]
assert decoupage(lcomptine,2, 6, 1) == lcomptine[2:6:1]
assert decoupage(lcomptine, 5, 2, -2) == lcomptine[5:2:-2]
assert decoupage(lcomptine, 6, 0, -1) == lcomptine[6:0:-1]
assert decoupage(lcomptine, 6, 0, -3) == lcomptine[6:0:-3]
```

Solution de l'exercice 7.1

Solution de la question 1

```python
def partie_reelle(c : Complexe) -> float:
    """Renvoie la partie réelle du nombre complexe c.
    """
    re, _ = c
    return re
```

```python
# Jeu de tests
assert partie_reelle((2.0,3.0)) == 2.0
assert partie_reelle((0.0,1.0)) == 0.0
assert partie_reelle((4.0,0.0)) == 4.0
```

```python
def partie_imaginaire(c : Complexe) -> float:
    """Renvoie la partie imaginaire du nombre complexe c.
    """
    _, im = c
    return im
```

```python
# Jeu de tests
assert partie_imaginaire((2.0,3.0)) == 3.0
assert partie_imaginaire((0.0,1.0)) == 1.0
assert partie_imaginaire((4.0,0.0)) == 0.0
```

Solution de la question 2

```python
def addition_complexe(c1 : Complexe, c2 : Complexe) -> Complexe:
    """Renvoie le nombre complexe correspondant à c1 + c2.
    """
    re1, im1 = c1
    re2, im2 = c2
    return (re1 + re2 , im1 + im2)
```

```python
# Jeu de tests
assert addition_complexe((1.0, 0.0), (0.0, 1.0)) == (1.0, 1.0)
assert addition_complexe((2.0, 3.0), (0.0, 1.0)) == (2.0, 4.0)
```

Solution de la question 3

```python
def produit_complexe(c1 : Complexe, c2 : Complexe) -> Complexe:
    """Renvoie le nombre complexe correspondant à c1 * c2.
    """
    re1, im1 = c1
    re2, im2 = c2
    return (re1*re2 - im1 * im2, re1 * im2 + im1 * re2)
```

```python
# Jeu de tests
assert produit_complexe((0.0, 0.0), (1.0, 1.0)) == (0.0, 0.0)
assert produit_complexe((0.0, 1.0), (0.0, 1.0)) == (-1.0, 0.0)
```

Solution de l'exercice 7.2

Solution de la question

```python
def nb_de_max(l : List[float]) -> Tuple[float, int]:
    """Précondition : len(l) > 0
    Renvoie le couple (m,n) dans lequel m est le maxumum de l
    et n le nombre d'occurrences de m dans l.
```

```python
    """
    # Candidat maximum
    m : float = l[0]

    # Nombre de fois où m est apparu dans l
    n : int = 1

    x : float
    for x in l[1:]:
        if x > m : # on a trouvé un nombre supérieur au maximum courant
            m = x
            n = 1
        elif x == m : # on a trouvé une nouvelle occurrence du maximum courant
            n = n + 1

    return (m, n)
```

```python
# Jeu de tests
assert nb_de_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5]) == (9, 2)
assert nb_de_max([-2, -1, -5, -3, -1, -4, -1]) == (-1, 3)
```

Solution de l'exercice 7.3

Solution de la question 1

```python
>>> sauvegarde_fichier("haiku.txt", ["Papillon voltige"
                                    ,"Dans un monde"
                                    , "Sans espoir."
                                    , "(Kobayashi Issa)"])
```

```python
>>> chargement_fichier("haiku.txt")
['Papillon voltige', 'Dans un monde', 'Sans espoir.', '(Kobayashi Issa)']
```

Solution de la question 2

```python
def decoupage_mots(phrase : str) -> List[str]:
    """Précondition : phrase est composée de mots séparés par des espaces
    Renvoie la liste des mots de la phrase.
    """
    # La liste des mots
    lmots : List[str] = []

    # Le mot courant
    mot : str = ""

    ch : str # Caractère courant dans la phrase
    for ch in phrase:
        if ch == ' ':
            if mot != "":
```

```python
                lmots.append(mot)
                mot = ""
        else:
            mot = mot + ch

    if mot != "":
        lmots.append(mot)

    return lmots
```

```python
# Jeu de tests
assert decoupage_mots("Dans un monde") == ["Dans", "un", "monde"]
assert decoupage_mots("Bonjour    Hello     ") == ["Bonjour", "Hello"]
assert decoupage_mots("") == []
```

Solution de la question 3

```python
def lecture_produit(ligne : str) -> Tuple[str, int, float]:
    """Précondition : la ligne de texte décrit une commande de produit.
    Renvoie la commande produit (nom, quantité, prix unitaire).
    """
    lmots : List[str] = decoupage_mots(ligne)
    nom_produit : str = lmots[0]
    quantite : int = int(lmots[1])
    prix_unitaire : float = float(lmots[2])
    return (nom_produit, quantite, prix_unitaire)
```

```python
# Jeu de tests
assert lecture_produit("Lait  12 2.0") == ('Lait', 12, 2.0)
assert lecture_produit("Tomate 6   1.5") == ('Tomate', 6, 1.5)
```

Solution de la question 4

```python
def lecture_commande(commande : List[str]) -> List[Tuple[str, int, float]]:
    """Précondition : la commande est dans le bon format
    Renvoie la liste des produits commandés.
    """
    lproduits : List[Tuple[str, int, float]] = []

    produit : str
    for produit in commande:
        lproduits.append(lecture_produit(produit))

    return lproduits
```

```python
>>> lecture_commande(chargement_fichier("commande.txt"))
[('Lait', 12, 2.0), ('Thé', 8, 4.5), ('Tomate', 6, 1.5), ('Fromage', 9, 8.5)]
```

Solution de la question 5

```python
def gen_facture(lproduits : List[Tuple[str, int, float]]) -> List[str]:
    """Précondition : lproduits est une liste de produits commandés.
```

```python
    Renvoie une facture éditée sous la forme d'une liste
    de lignes de texte.
    """
    lfacture : List[str] = ["Produit  Prix",
                            "-------  ----"]

    # Prix total
    total_ht : float = 0.0

    nom_produit : str
    quantite : int
    prix_unitaire : float
    for (nom_produit, quantite, prix_unitaire) in lproduits:
        lfacture.append(nom_produit + "  " + str(quantite * prix_unitaire))
        total_ht = total_ht + quantite * prix_unitaire

    lfacture.append("")  # ajout d'une ligne vide
    lfacture.append("Total_HT    " + str(total_ht))

    lfacture.append("TVA_20%     " + str(total_ht * 20.0 / 100.0))
    lfacture.append("Total_TTC   " + str(total_ht + (total_ht * 20.0 / 100.0)))

    return lfacture
```

```python
>>> sauvegarde_fichier('facture.txt',
                       gen_facture(lecture_commande(
                                   chargement_fichier('commande.txt'))))
```

Solution de l'exercice 8.1

Solution de la question 1

```python
def repetition(x : T, k : int) -> List[T]:
    """Précondition : k >= 0
    Retourne la liste composée de k occurrences de x.
    """
    return [x for i in range(1, k + 1)]
```

```python
# Jeu de tests
assert repetition(0, 4) == [0, 0, 0, 0]
assert repetition(4, 0) == []
assert repetition('pom', 5) == ['pom', 'pom', 'pom', 'pom', 'pom']
```

Solution de la question 2

```python
def liste_diviseurs(a : int) -> List[int]:
    """Précondition : a > 0
    Retourne la liste des diviseurs de a.
    """
```

```python
    return [i for i in range(1, a + 1) if a % i == 0]
```

```python
# Jeu de tests
assert liste_diviseurs(2) == [1, 2]
assert liste_diviseurs(12) == [1, 2, 3, 4, 6, 12]
assert liste_diviseurs(25) == [1, 5, 25]
```

```python
def liste_diviseurs_impairs(a : int) -> List[int]:
    """Précondition : a > 0
    Retourne la liste des diviseurs impairs de a.
    """
    return [i for i in range(1, a + 1)
              if (a % i == 0) and (i % 2 == 1)]
```

```python
# Jeu de tests
assert liste_diviseurs_impairs(2) == [1]
assert liste_diviseurs_impairs(12) == [1, 3]
assert liste_diviseurs_impairs(30) == [1, 3, 5, 15]
assert liste_diviseurs_impairs(15) == [1, 3, 5, 15]
```

Solution de l'exercice 8.2

Solution de la question 1

```python
def alphabet() -> List[str]:
    """Retourne la liste des 26 lettres de l'alphabet.
    """
    return [chr(i) for i in range(ord('a'), ord('z') + 1)]
```

Solution de la question 2

```python
def est_voyelle(c : str) -> bool:
    """Précondition : len(c) == 1
    Retourne True si c est une voyelle, ou False sinon.
    """
    return (c == 'a') or (c == 'e') or (c == 'i') or (c == 'o') \
           or (c == 'u')  or (c == 'y')
```

```python
# Jeu de tests
assert est_voyelle('a') == True
assert est_voyelle('c') == False
assert est_voyelle('e') == True
assert est_voyelle('g') == False
```

Solution de la question 3

```python
>>> [c for c in alphabet() if est_voyelle(c)]
['a', 'e', 'i', 'o', 'u', 'y']
```

Solution de la question 4

```
>>> [c for c in alphabet() if not est_voyelle(c)]
['b',
 'c',
 'd',
 'f',
 'g',
 'h',
 'j',
 'k',
 'l',
 'm',
 'n',
 'p',
 'q',
 'r',
 's',
 't',
 'v',
 'w',
 'x',
 'z']
```

Solution de l'exercice 8.3

Solution de la question 1

```
def liste_non_multiple(n : int, l : List[int]) -> List[int]:
    """Précondition : n != 0
    Renvoie la liste des éléments de L qui ne sont pas multiples de n.
    """
    return [e for e in l if e % n != 0]
```

```
# Jeu de tests
assert liste_non_multiple(2,[2,3,4,5]) == [3,5]
assert liste_non_multiple(2,[2,4,6]) == []
assert liste_non_multiple(3,[2,3,4,5]) == [2,4,5]
assert liste_non_multiple(2,[]) == []
assert liste_non_multiple(7,[2,3,4,5]) == [2,3,4,5]
```

Solution de la question 2

```
def eratosthene(n : int) -> List[int]:
    """Précondition : n > 1
    Renvoie la liste des nombres premiers inférieurs ou égaux à n.
    """
    # Liste de départ
    l : List[int] = [ k for k in range(2,n+1)]

    # Liste contenant les nombres premiers
```

```python
    lp : List[int] = []

    # Prochain nombre premier
    p : int =0 # prochain nombre premier

    while len(l) > 0:
        p = l[0] # on récupère le prochain nombre premier
        lp.append(p) # on le rajoute à la liste courante
        # puis on calcule les entiers non multiples du nombre premier
        l = liste_non_multiple(p, l)
    return lp
```

```python
# Jeu de tests
assert eratosthene(10) == [2,3,5,7]
assert eratosthene(2) == [2]
assert eratosthene(40) == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Solution de la question 3

```python
def liste_facteurs_premiers(n : int) -> List[int]:
    """Précondition : n > 1
    Renvoie la liste des facteurs premiers de n.
    """
    return [e for e in eratosthene(n) if n%e == 0]
```

```python
# Jeu de tests
assert liste_facteurs_premiers(2) == [2]
assert liste_facteurs_premiers(10) == [2, 5]
assert liste_facteurs_premiers(2*3*7) == [2, 3, 7]
assert liste_facteurs_premiers(2*3*4*7*9) == [2, 3, 7]
```