Modularité

&

Mise au point des programmes

Cours

Capacités attendues	
Modularité	▪ Utiliser des API (Application Programming Interface) ou des bibliothèques. ▪ Exploiter leur documentation. ▪ Créer des modules simples et les documenter.
Mise au point des Programmes. Gestion des bugs.	▪ Savoir répondre aux causes typiques de bugs : problèmes liés au typage, effets de bord non désirés, débordements dans les tableaux, instruction conditionnelle non exhaustive, choix des inégalités, comparaisons et calculs entre flottants, mauvais nommage des variables, etc.

1. Modularité

Lorsque le code d'un programme dépasse quelques centaines de lignes, il devient difficile de le mettre au point, c'est-à-dire de corriger ses bugs, et de le maintenir (évolutions).

Il est alors indispensable de le découper en plusieurs composants aussi indépendants que possible les uns des autres.

Ces composants sont appelés **modules** et un programme ainsi découpé en modules est dit **modulaire**. On peut alors tester chaque module indépendamment, c'est le **test unitaire**, puis tester les modules ensemble, c'est le **test d'intégration**.

La modularité facilite également **la réutilisation** : un module bien conçu peut être utilisé par d'autres programmes et mis à disposition de la communauté dans la communauté dans une **bibliothèque** de modules.

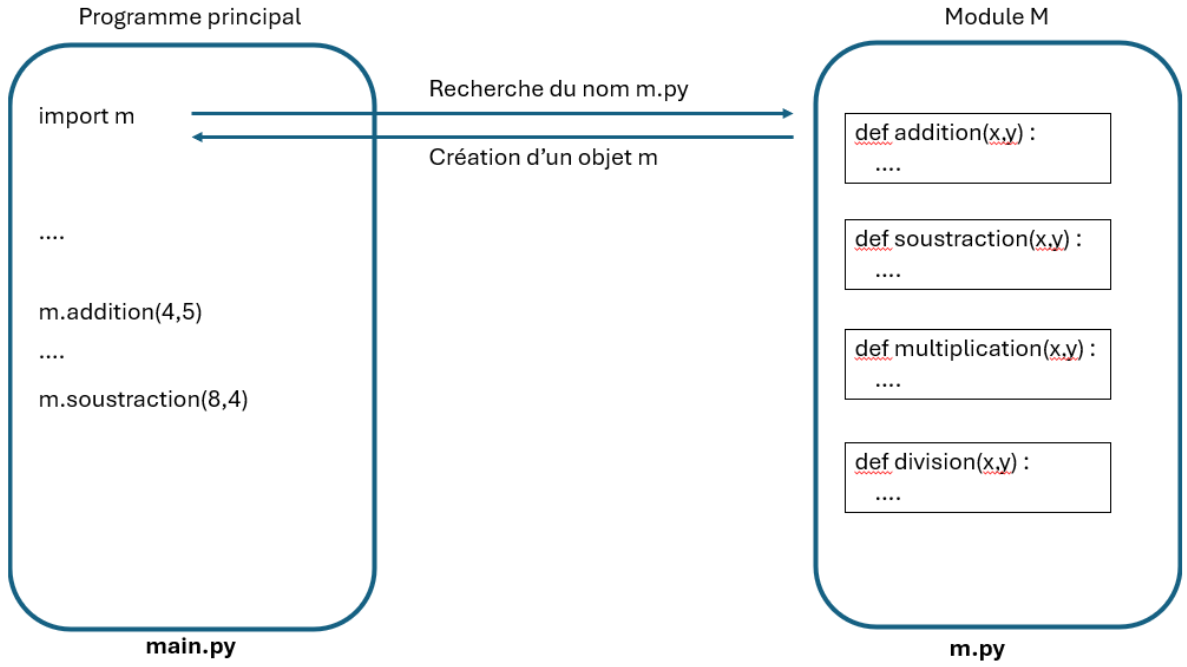
1.1 Créer un module

Un module définit un ensemble de fonctions et parfois de variables globales que l'on peut utiliser dans le programme principal ou dans un autre module sans avoir à connaître le détail de leur implémentation.

Une variable globale est une variable déclarée à l'extérieur du corps de toute fonction, et pouvant donc être utilisée n'importe où dans le programme. On parle également de **variable de portée globale**.

L'ensemble de ces définitions constitue **l'interface de programmation** du module ou **API** (*Application Programming Interface*). On dit que le module **exporte** ces définitions. Un programme (ou un module) **importe** un autre module pour utiliser ses définitions.

En Python, un module correspond à un fichier contenant l'ensemble du code nécessaire à l'implémentation de son API. L'instruction Python **import m** recherche un fichier de nom **m.py** dans le même répertoire que le fichier contenant l'instruction **import**, puis dans un ensemble de répertoires standards contenant les bibliothèques installées sur l'ordinateur. Si le fichier est trouvé, l'instruction créer un objet **m** dont les attributs sont les fonctions et variables exportées par le module, que l'on référence par la notion pointée **m.f**



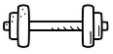
1.2 Utiliser des bibliothèques de modules

Une **bibliothèque** est un ensemble d'un ou plusieurs modules qui sont installés sur l'ordinateur et qui permettent d'utiliser des fonctionnalités sans avoir besoin de les programmer soi-même. Python inclut une **bibliothèque standard** avec des modules tels que **math** ou **random**.

Il existe de nombreuses bibliothèques que l'on peut installer sur son ordinateur en plus de la bibliothèque standard. Ces bibliothèques sont recensées dans des archives dont la plus connue est https://pypi.org

Matplotlib (Affichage de graphiques), **PIL** (manipulation d'images), **numpy** (calcul numérique) et **pandas** (manipulation de données) sont des exemples de bibliothèques très populaires.

Utiliser des bibliothèques permet de réduire le coût de développement car cela permet de réutiliser le travail fait par d'autres. Les bibliothèques les plus répandues sont robustes car elles ont été testées par de nombreux utilisateurs. Elles sont aussi souvent optimisées pour être les plus efficaces possibles.

 **Activité :** A l'aide de la documentation Turtle :
https://docs.python.org/3/library/turtle.html
Et des instructions du fichier ci-dessous, afficher une étoile.

```
import turtle

# Initialiser l'ecran
screen = turtle.Screen()
screen.title("Dessin d'une Étoile avec Turtle")

# Créer une tortue
star_turtle = turtle.Turtle()
star_turtle.XXXXX  # Regler la vitesse de la tortue

# Dessiner une etoile
for _ in range(5):
    star_turtle.XXXXXXX # Avancer de 100 unités
    star_turtle.XXXXXXX)    # Tourner de 144 degrés pour former une etoile

# Fermer la fenetre au clic
screen.exitonclick()
```

1.3 Spécifier et documenter un module

Pour pouvoir utiliser un module sans connaître le détail de son implémentation, l'auteur du module doit **spécifier** les fonctions et variables qu'il définit :

- La spécification d'une fonction décrit ce qu'elle fait et donne la liste de ces paramètres et leurs types, ainsi que le type de la valeur de retour ;
- La spécification d'une variable globale décrit son type et son rôle.

En python on peut documenter chaque fonction et module par une chaîne de caractères appelée *docstring*, placée juste après sa définition.

Exemple de docstring pour une fonction :

```
def addition(a, b):
    """
    Ajoute deux nombres et retourne le resultat.

    Args:
        a (int ou float): Le premier nombre a additionner.
        b (int ou float): Le second nombre a additionner.

    Returns:
        int ou float: La somme de `a` et `b`.
    """
    return a + b
```

Exemple de docstring pour une variable :

En Python, il n'y a pas de docstring formelle pour les variables comme pour les fonctions, mais on peut utiliser un commentaire juste au-dessus de la variable pour la décrire.

```
# Le taux d'intérêt annuel pour les calculs financiers.
taux_interet_annuel = 0.05
```

Exemple de docstring pour un module:

En Python, il est possible d'ajouter des docstrings aux modules. La docstring d'un module est généralement placée au tout début du fichier, avant toute autre instruction (à part éventuellement les commentaires ou les encodages de fichiers).

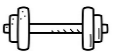
```
"""
Ce module fournit des fonctions pour effectuer des opérations mathématiques avancées.
Il inclut des fonctions pour l'intégration, la dérivation, et les transformations de Fourier.

Exemples d'utilisation:
  from mon_module import integration
  result = integration(fonction, borne_inf, borne_sup)

Auteur: John Doe
Date: 2024-09-19
"""

def integration(fonction, borne_inf, borne_sup):
  pass

def derivation(fonction, point):
  pass
```

Accéder à la docstring d'un module

Une fois que le module est importé, vous pouvez accéder à sa docstring en utilisant l'attribut __doc__ :

```
import mon_module
print(mon_module.__doc__)
```

Il est aussi possible, en mode interactif d'utiliser la fonction help(x) ou x est le nom de la fonction.

 **Activité :** Faire les exercices « docstring » puis « modularité ».

1.4 Le type hinting

Le type hinting en Python est une fonctionnalité qui permet d'indiquer, à travers des annotations, le type de données que les variables, les arguments de fonction ou les valeurs de retour d'une fonction sont censés prendre. C'est une aide pour le développeur, mais cela n'est pas vérifié à l'exécution par l'interpréteur Python lui-même.

Pourquoi utiliser le type hinting ?

1. **Lisibilité** : Cela rend le code plus lisible, car on sait à l'avance quel type est attendu pour une variable ou une fonction.
2. **Détection d'erreurs** : Cela permet aux outils d'analyse de code de détecter des erreurs potentielles avant même l'exécution du programme.
3. **Documentation** : C'est une forme implicite de documentation qui aide à comprendre l'intention derrière le code.

Voici quelques exemples :

1. Typage des arguments et de la valeur de retour d'une fonction

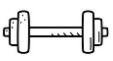
```
def add(a: int, b: int) -> int:
    return a + b
```

2. Typage des variables

```
age: int = 25
name: str = "Alice"
```

3. Typage des listes et des collections

```
def process(items : list[int]) -> None:
  for item in items:
    print(item)
```

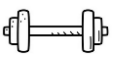
 **Activité :** Faire les exercices « type hinting ».

3.2 Les préconditions et les postconditions

Les préconditions et postconditions sont des assertions qui définissent ce qui doit être vrai avant l'exécution d'une fonction (préconditions) et ce qui doit être vrai après son exécution (postconditions). En Python, on peut utiliser des assertions pour vérifier ces conditions, généralement à des fins de débogage.

Utilisation
Les assertions (assert) permettent de vérifier les préconditions et les postconditions pendant l'exécution du code. Si une de ces assertions échoue, une exception *AssertionError* est levée, indiquant que la condition spécifiée n'a pas été respectée.

```
def division(a, b):
    """
    Divise a par b.

    Préconditions:
    - b ne doit pas être égal à 0.
    - a et b doivent être des nombres (int ou float).

    Postconditions:
    - Le résultat doit être un nombre (float ou int).

    :param a: Le numérateur.
    :param b: Le dénominateur.
    :return: Le résultat de la division de a par b.
    """
    # Préconditions
    assert isinstance(a, (int, float)), "Le numérateur (a) doit être un nombre (int ou float)."
    assert isinstance(b, (int, float)), "Le dénominateur (b) doit être un nombre (int ou float)."
    assert b != 0, "Le dénominateur (b) ne doit pas être égal à 0."

    # Calcul de la division
    result = a / b

    # Postconditions
    assert isinstance(result, (int, float)), "Le résultat doit être un nombre (int ou float)."

    return result
```



Activité : Faire l'exercice préconditions et postconditions.

3.3 Les différentes techniques d'import de module

En Python, il existe plusieurs façons d'importer des modules. Chaque méthode a ses avantages et ses inconvénients, en fonction de l'utilisation prévue. Voici les principales méthodes d'importation en Python :

a) Importation simple

import module_name

Exemple :

```
import math

print(math.sqrt(16))
```

Avantages :

- Clair et explicite : il est facile de voir d'où proviennent les fonctions utilisées.
- Pas de risque de conflit de noms, car les objets doivent être appelés avec le préfixe du module (par exemple, math.sqrt).

Inconvénients :

- Peut-être verbeux si le nom du module est long ou si de nombreuses fonctions sont utilisées fréquemment.

b) Importation avec un alias

import module_name as alias

Exemple :

```
import numpy as np

array = np.array([1, 2, 3])
```

Avantages :

- Réduit la verbosité : les modules avec des noms longs peuvent être abrégés (comme numpy en np).
- Toujours explicite, car l'alias indique clairement d'où proviennent les objets.

Inconvénients :

- Peut-être confus si l'alias est trop vague ou peu intuitif.

c) Importation de composants spécifiques

from module_name import component

Exemple :

```
from math import sqrt

print(sqrt(16))
```

Avantages :

- Rend le code plus concis en évitant le préfixe du module.
- Peut améliorer les performances si seuls quelques composants d'un module volumineux sont nécessaires.

Inconvénients :

- Risque de conflits de noms si le même nom est défini dans plusieurs modules ou dans le code.

d) Importation de tous les composants

from module_name import *

Exemple :

```
from math import *

print(sqrt(16))
```

Avantages :

- Très concis, car tous les composants du module sont importés.
- Utile pour de petites applications pour tester rapidement des fonctionnalités.

Inconvénients :

- Risque élevé de conflits de noms, surtout avec des modules ayant de nombreuses fonctions.

4. Tester et mettre au point

Lorsque l'on exécute un programme il peut fonctionner comme prévu, mais il peut aussi « planter » ou bien boucler indéfiniment, ou bien ne pas produire le résultat escompté. Et même s'il fonctionne comme prévu, rien ne garantit que ce soit toujours le cas quelles que soient les données d'entrée qu'on lui fournit.

Tester un programme (pour une fonction, un module) consiste à vérifier qu'il fonctionne comme prévu dans un maximum de situations possibles. Un **jeu de test** est un programme ou un ensemble de programmes qui essaient de mettre en défaut la partie testée. Si un test échoue il faut **mettre au point** le code, c'est-à-dire identifier puis corriger le problème, et effectuer à nouveau le test.

2.1 Plusieurs types de tests

On utilise plusieurs types de tests à différentes étapes du développement.

On commence par des **tests unitaires**, qui concernent une petite unité d'un programme typiquement une fonction.

On effectue ensuite **des tests d'intégration**, qui consistent à vérifier que deux parties ou plus d'un programme fonctionnent correctement ensemble. Par exemple, on teste séparément (tests unitaire) une partie qui obtient des données d'un site Web, une autre qui réalise des calculs sur ces données et une troisième qui les affiche sous forme graphique. Puis on teste l'ensemble du programme (test d'intégration).

On peut être amené à effectuer des **tests de performance** pour vérifier que le logiciel se comporte correctement lorsqu'il est confronté à des données de grande taille ou pour un serveur, à un grand nombre de connexions.

Enfin, les **tests d'utilisabilité** évaluent l'ergonomie du programme, c'est-à-dire la capacité des utilisateurs à effectuer les tâches souhaitées avec le programme de manière simple et efficace, grâce à une interface appropriée.

Les tests unitaires et d'intégration se divisent en **tests fonctionnels**, qui vérifient qu'un programme ou une partie du programme (par exemple une fonction) est bien conforme à sa spécification, et **tests structurels**, qui vérifie le fonctionnement interne du programme.

Les tests fonctionnels sont souvent appelés **tests « boîtes noires »** car ils sont définis à partir de la *spécification* de l'entité testée, sans connaissance de son *implémentation*.

Une **spécification** est un ensemble explicite d'exigences à satisfaire.
L'implémentation, c'est la **mise en œuvre** technique d'un programme (le code) à partir d'un document de spécification.

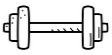
A l'inverse, les tests structurels sont aussi appelés **tests « boîtes blanches »** car ils sont écrits

en fonction de *l'implémentation* du programme et cherchent à couvrir les différents chemins d'exécution : les branches d'une conditionnelle, les cas limites d'une boucle, etc ...

 **Activité :** Faire correspondre chaque type de test à sa définition.

	Les tests
a	Unitaires
b	D'intégration
c	Fonctionnels (boîte noire)
d	Structurels (boîte blanche)
e	De performance
f	D'utilisabilité

	Permettent de valider
1	Plusieurs parties du programme ensemble
2	Par rapport à la spécification
3	Une partie du programme
4	L'ergonomie du programme
5	Par rapport à la taille des données
6	Par rapport à l'implémentation

 **Activité :** Faire les exercices tests « boite blanche » et tests « boite noire ».

2.2 Corriger les erreurs

La **mise au point** d'un programme consiste à corriger les erreurs identifiées par les tests. Certaines erreurs sont faciles à corriger lorsque le message d'erreur ou le test indique clairement l'origine de l'erreur. D'autres sont plus difficiles à traquer et nécessitent un véritable travail de détective pour *localiser* l'erreur, *identifier* sa cause et enfin la *corriger*. Les erreurs les plus difficiles à corriger sont celles pour lesquelles la cause de l'erreur est très éloignée de l'endroit du programme où elle est détectée.

L'approche classique pour localiser une erreur et identifier sa cause et d'ajouter des **traces** dans le programme, à l'aide d'instructions **print**. L'instruction **assert** est également utile pour arrêter l'exécution dès que l'on détecte un résultat incorrect, par exemple un *invariant* qui n'a plus satisfait. Lorsque l'assertion est fausse, python affiche la pile des appels de fonctions, ce qui permet de voir la chaîne d'appel qui a conduit à l'erreur.
Une autre approche pour comprendre les causes d'une erreur est d'utiliser un **outil de débogage**. Un tel outil permet d'exécuter un programme « pas-à-pas », c'est-à-dire instruction par instruction, d'afficher les valeurs des variables à chaque pas et de poser des *points d'arrêt*, c'est-à-dire des instructions sur lesquelles le programme s'arrêtera. Il permet ainsi de suivre l'exécution du programme « à la trace ».
Une fois l'erreur corrigée, il faut s'assurer que l'on n'a pas créé de nouveaux problèmes en exécutant à nouveau *tous* les tests, pas seulement ceux qui ont causé l'erreur.

Un variant de boucle est une méthode utilisée en programmation pour s'assurer qu'une boucle va se terminer. Elle consiste en une valeur numérique ou une condition qui change à chaque itération de la boucle, de manière à se rapprocher d'une situation où la boucle peut s'arrêter.

Un invariant est une propriété qui reste toujours vraie à chaque étape d'une boucle, du début à la fin. Il est utilisé pour s'assurer que la boucle fonctionne correctement et pour prouver que l'algorithme est correct.

Exemple : Pour le tri par sélection, à chaque itération, les éléments avant la position courante sont triés et sont les plus petits éléments de la liste.

5. Types d'erreurs

Les sources possibles d'erreurs dans un programme sont nombreuses. La suite de cette section décrit les types d'erreurs les plus fréquentes et donne des indications sur la manière de les détecter et de les corriger. Pour réduire les risques d'erreurs, il est recommandé d'adopter une approche de **programmation défensive**, c'est à dire d'anticiper les erreurs possibles en prenant en compte les indications ci-dessous dès la programmation.

3.1 Erreur de syntaxe

Les erreurs de syntaxe sont dues à des instructions mal formées, qui ne suivent pas les règles du langage. Elles sont détectées dès le début de l'exécution du programme et sont en général faciles à corriger car le message d'erreur est explicite.

En python, une erreur fréquente est une mauvaise indentation du code ou le mélange d'espaces et de tabulations. Une autre erreur fréquente et vous oublie de **:** à la fin des instructions telles que **def** , **if** ou **while** , ou l'utilisation de **=** au lieu de **==** pour un test d'égalité.

3.2 Erreur de type

Contrairement au langage typé tels que Java, le langage Python ne permet pas de déclarer les types attendus des variables et des paramètres et ne peut donc pas les contrôler avant l'exécution du programme.

Pour se protéger des erreurs de type, il faut utiliser des pré-conditions au début des fonctions pour valider les types des paramètres, notamment avec la fonction **isinstance(val, t)**, qui retourne vrai si **val** et de type **t**.
t peut-être un des types de Python (**int**, **float**, **str**, **tuple**, **list**, **dict**).

Exemple d'erreur de type :

```
def f(x) :
    return x * x

f("toto")
```

3.3 Erreur d'accès

Python vérifie la validité des accès à un élément de tableau ou de tuple, à une clé d'un dictionnaire et à un attribut ou une méthode d'un objet seulement à l'exécution du programme. Pour se protéger de ces erreurs, on peut utiliser les tests suivants :

Expression	Valide si	Test
t[i]	i est un indice du tableau t	i>=0 and i < len(t)
a, b = t	t est un tuple de 2 éléments	len(t) == 2
d[k]	k est une clé du dictionnaire	k in d

3.4 Effets de bord

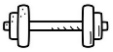
Un effet de bord est une modification d'une variable qui affecte l'état du programme en dehors de la fonction ou de la méthode où elle a lieu. Un Python cela arrive principalement de deux manières : par les variables globales et par l'aliassage.

Une variable globale est une variable déclarée en dehors de toute fonction. En Python, elle peut être utilisée partout dans le programme. En revanche, elle ne peut être modifiée dans une fonction que si on l'a déclarée par l'instruction **globale x** dans cette fonction. Par conséquent, il est rare de créer un effet de bord ainsi par erreur.

Exemple :

a = 14
 	
def x(x, y) :
  return a + x + y
 	
print(x(23,2))
	a = 14
	
	def x(x, y) :
	  a = 1
	  a = a + x + y
	  return a
	
	print(x(23,2))
	print(a)
		a = 14
		
		def x(x, y) :
		  global a
		  a = a + x + y
		  return a
		
		print(x(23,2))
		print(a)

?	? ?	? ?

 **Activité :** Pour chaque exemple ci-dessus, à quelle ligne la variable *a* est une variable locale, une variable globale ? Quand la variable *a* est utilisée, modifiée ?

Exemple avec redéfinition d'une variable locale :

```
a = 14

def x(x, y) :
    a = 0
    a = a + x + y
    return a


print(x(23,2))
print(a)
```

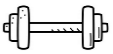
Ici, la variable a est redéfinie comme variable locale à la fonction.

```
>>> %Run -c $EDITOR_CONTENT
 25
 14
```

L'aliassage survient lorsque deux variables référencent la même donnée, ce qui peut arriver avec les types structurés tableaux et dictionnaires notamment lorsqu'on les passe en paramètre.

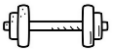
Exemple d'aliassage :

```
t1 = ['a']
t2 = t1

t2[0] = 'b'
print(t1)

def f(t):
    t[0] = 'z'
f(t1)

print(t1)
print(t2)
```

 **Activité :** A l'aide de Python Tutor (https://pythontutor.com), exécuter le programme ci-dessus.
Expliquer chaque instruction et leurs conséquences.

Les effets de bord indésirables sont particulièrement difficiles à détecter car, en général, ils ne provoquent pas d'erreur d'exécution et ils se manifestent en d'autres points du code que leur origine.

Pour limiter les risques, il est important de bien documenter chaque fonction. Si une fonction modifie un paramètre contenant une valeur mutable, il faut l'indiquer explicitement dans la spécification ou bien éviter l'effet de bord **en effectuant une copie de la valeur**.

 **Activité :** Effectuez en premier les exercices spécifiques « effets de bords » puis « exercices ».

3.5 Conditionnelles

Les instructions conditionnelles **if** comportent des expressions booléennes qui déterminent le flot d'exécution du programme. Les principaux types d'erreurs sont les suivants :

- Oubli de certains cas lorsque l'on évalue plusieurs alternatives avec **elif** ;
- Mauvaises conditions aux limites lorsque l'on compare une valeur à une borne, par exemple en utilisant **<** au lieu de **<=** ou l'inverse ;
- Mauvais usage des opérateurs booléens et en particulier de l'évaluation « court-circuit » : **i < n and t[i] == 0** est différent de **t[i] == 0 and i < n**, qui provoque une erreur si **i** est en dehors des bornes de **t**.

Pour éviter ces erreurs, il faut effectuer des tests qui couvrent les différents cas : s'assurer que toutes les branches donc conditionnelle sont testées, ainsi que les cas limites des expressions booléennes.

3.6 Boucles

Les boucles bornées **for** et non bornées **while** contrôlent également le flot d'exécution du programme.
Les boucles non bornées sont contrôlées par une expression booléenne, aussi les mêmes erreurs de mauvaises conditions aux limites et de mauvais usage des opérateurs booléens que ci-dessus peuvent se produire. Ces erreurs peuvent conduire à des boucles infinies, mais la raison la plus courante d'une boucle infinie est de ne pas modifier correctement dans le corps de la boucle les valeurs qui sont utilisées dans la condition booléenne de la boucle.

Il est important d'identifier le *variant de boucle,* l'expression qu'il doit changer à chaque itération de la boucle de telle sorte que la condition deviendra fausse.

Les boucles bornées ne peuvent pas produire de boucles infinies. L'erreur la plus fréquente est la mauvaise évaluation des bornes en particulier avec les expressions **range(n)** et **range(i, j)** qui énumèrent respectivement les valeurs de 0 à n-1.

Comme pour les conditionnelles, il est important de tester son code en prenant soin de couvrir tous les cas limites. Les assertions peuvent également aider à la mise au point, par exemple pour tester un variant de boucle.

Note :
L'utilisation de la récursion peut, comme la boucle non bornée donnée lieu à une exécution infinie si l'on ne traite pas ou incorrectement le cas de base qui arrête la récursion ou si l'appel récursif n'atteint jamais le cas de base.

3.7 Calcule en virgule flottante

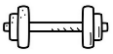
Les calculs en virgule flottants sont source d'erreurs car un ordinateur ne peut pas représenter de manière exacte tous les nombres réels. Par exemple en python 0.1+0.2 vaut :
0.3 0000000000000004.
Un problème similaire se manifeste lorsqu'on ajoute ou soustrait un nombre très grand et un nombre très petit 10^e20 -1 , 10^e20 , 10^e20 + 1 sont égaux en Python.

Il en résulte qu'il ne faut jamais utiliser des tests d'égalité stricts sur les nombres flottants mais la fonction **isclose(a,b)** de la bibliothèque **math** :

```
from math import isclose

print(0.1 + 0.2 == 0.3, isclose(0.1 + 0.2, 0.3))
```

Une autre source de problèmes dus à ces approximations est l'accumulation d'erreurs. Dans l'exemple suivant, on voit qu'il est préférable de faire une multiplication plutôt qu'une suite d'additions :

```
s = 0.0
for i in range(1000) :
   s += 0.1
print(s, 0.1 * 1000)
```

 **Activité :** Exécutez le code Python de la section ci-dessus.

3.8 Nommage des variables

La lisibilité d'un programme est grandement facilitée si on donne des noms expressifs aux fonctions, variable, ... Des noms bien choisis rendent le code plus facile à comprendre et donc les erreurs plus faciles à détecter. Ils réduisent aussi le risque de *masquage* de nom qui peut être une erreur difficile à identifier. En Python il y a deux situations où un nom peut être masqué par un autre :

- On importe une bibliothèque et on déclare localement une fonction qui a le même nom qu'une fonction importée. Ce risque est particulièrement élevé si on importe toutes les fonctions de bibliothèque avec **from m import *** , car on ne sait pas quels noms sont ainsi importés ;
- On définit une fonction avec un paramètre qui a le même nom qu'une variable globale. A l'intérieur de cette fonction le paramètre masque la variable globale qui ne peut être

utilisée.

Pour éviter de masquer un nom exporté par un module, Il est recommandé d'utiliser l'instruction **import m** qui oblige à qualifier tous les noms de **m** avec l'annotation pointée **m.f** Cela a l'avantage de bien rendre visible les utilisations du module **m**.

Pour les variables globales, la meilleure approche est de leur donner des noms explicites, par exemple qui commencent par **g_**.
Par ailleurs les variables globales d'un programme sont souvent en réalité des constantes. La convention en python est de leur donner des noms entièrement en majuscules.

Une constante est une valeur fixe qui ne change pas pendant l'exécution d'un programme. Contrairement à une variable, qui peut être modifiée, une constante reste la même tout au long du programme.

6. Construire et exécuter des jeux de tests

Un **jeu de tests** est un ensemble de tests destinés à valider un aspect du programme. Par exemple, un jeu de test unitaire d'une fonction inclut l'ensemble des tests « boîte noire » et « boîte blanche » de cette fonction. Un jeu de tests est constitué d'un ou plusieurs programmes qui importent ou incluent la partie à tester et contiennent les différents tests.

Pour faciliter l'exécution d'un jeu de tests en Python, on peut utiliser pytest qui permet d'exécuter un ensemble de fonctions de test contenues dans un fichier.

Une **fonction de test** est une fonction dont le nom commence par **test_** ou termine par **_test**.
Elle doit contenir une assertion qui est vraie si le test est correct.

Il est de bonne pratique de fournir avec chaque module un module de test. Ainsi lorsque l'on modifie le module, on peut rapidement vérifier qu'il passe les tests avec succès : on parle de **non-régression**.