Introduction

Sources :

- *« Manuel de Terminale NSI » de Michel Baudouin-Lafon aux éditions Hachette.*
- *« Manuel de Terminale NSI » de T. Balabonski, S. Conchon, JC. Filliâtre, K.Nguyen aux éditions Ellipse.*
- *« Documents ressources Eduscol »*

  https://eduscol.education.fr/2068/programmes-et-ressources-en-numerique-et-sciences-informatiques-voie-g

1 Modularité

Objectif 1

En première on a présenté la programmation procédurale, qui consiste à découper le programme en fonctions. En terminale, on introduit la programmation objet qui permet de mieux organiser les grands programmes. À partir d'une certaine échelle, il est nécessaire de découper encore le code en modules qui vont contenir des fonctions ou des classes répondant aux mêmes problèmes. Ces modules peuvent être partagés avec la communauté de développeurs.

Définition 1 ***Module et interface***

- Lorsqu'un projet logiciel devient grand et complexe, il est difficile de le déboguer et de le faire évoluer d'autant plus si les composants sont interdépendants.

  Une bonne pratique de conception logicielle consiste à organiser le projet en **modules** qui rassemblent des unités fonctionnelles du programme.

- Chaque module peut être importé dans un autre module dit *client*.

- Le principe **d'encapsulation** distingue pour un module, son **interface** publique de son **implémentation** privée :

  - L'**interface** ou **Application Programming Interface** définit les fonctionnalités offertes par le module et leur mode d'utilisation. Cette interface est accessible dans la *documentation* du module où sont décrites :
    - chaque *constante* ou *variable globale* ;
    - la **spécification** (paramètres avec leurs types, description du traitement, valeur(s) de retour avec leur type) de chaque *fonction* offerte ;
    - la **spécification** (description des attributs, spécifications de chaque méthode) de chaque *classe* offerte.

    Il faut bien concevoir l'interface avant l'implémentation, car on doit pouvoir modifier l'implémentation sans mise à jour de l'interface pour les clients.

- L'**implémentation** désigne le code source du module, elle est *privée* et sa connaissance ne doit pas être nécessaire à l'utilisation du module par un client. Ainsi les développeurs du module peuvent la faire évoluer sans impact pour les clients.

☞ Un ensemble de modules constitue une **bibliothèque de modules**. On peut ainsi partager des outils logiciels avec la communauté comme par exemple la *bibliothèque standard de Python*.

Méthode *Modules en Python*

☞ Un **module** est simplement un programme enregistré dans un fichier d'extension `.py`, par exemple `module.py`.

☞ Un **module** peut être importé dans un module client avec l'instruction `import`. En pratique on importe les noms des objets définis dans le module importé. L'ordre d'import des modules est donc important et les imports sont définis en début de programme.

- Pour éviter les conflits de noms, la bonne pratique est d'importer `module.py` avec `import module`. Les noms d'objets définis dans `module` sont alors accessibles avec la *notation pointée* : `module.nom`

```python
# exemple d'import du module math
import math

def rayon_disque(aire):
    return math.sqrt(aire / math.pi)
```

- En prenant soin de ne pas masquer des noms déjà définis dans le client, on peut importer uniquement certains noms d'objets avec `from module import nom1, nom2`.

```python
# exemple d'import de deux noms du module math
from math import pi, sqrt

def rayon_disque(aire):
    return sqrt(aire / pi)
```

- On peut importer tous les noms d'objets définis avec : `from module import *`

  ⚠ *Cet import non contrôlé est une mauvaise pratique, susceptible de masquer les noms définis dans le client ou dans d'autres modules importés selon l'ordre d'import.*

```python
# exemple d'import de tous les noms du module math
def trunc(chemin):
    """Tronque l'extension de fichier dans un chemin"""
    champs = chemin.split('.')
    return champs[:-1]

from math import *
# la fonction trunc définie dans le client
# est masquée par la fonction trunc du module math
```

- 💡 On peut importer dans n'importe quel module client/script, les modules de la *bibliothèque standard de Python* et des bibliothèques installées dans certains répertoires connus de Python. Les modules définis en dehors des répertoires d'installation ne peuvent être importés que dans un module client situé dans le *répertoire de travail* (celui du module client/script par défaut). La liste des répertoires d'installation est accessible par `sys.path`.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3
    .8/lib-dynload', '/home/administrateur/.local/lib/python3.8/site
    -packages', '/usr/local/lib/python3.8/dist-packages']
```

- On peut afficher l'interface d'un module importé depuis une console interactive avec `help module`. Par exemple, si on a importé le module `math` de la bibliothèque standard, on affiche son interface avec `help(math)` (noter que le module n'est pas écrit en Python mais en langage C).

```
NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.9/library/math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS

    # extrait
    log(...)
        log(x, [base=math.e])
        Return the logarithm of x to the given base.

        If the base not specified, returns the natural logarithm (base
              e) of x.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /opt/anaconda3/lib/python3.9/lib-dynload/math.cpython-39-x86_64-
         linux-gnu.so
```

Exercice 1 *Concours de jeux vidéos partie 1*

On veut écrire une application de traitement des résultats d'un concours de jeu vidéo. Le projet est découpé en trois modules

- un module `table_csv.py` regroupant les fonctions d'import/export de tables de données (`email` et `score`) entre différents format (tableau de dictionnaires, fichiers `csv`) :

```
email,score
wmeyer@mercier.fr,6546239
prevostjerome@noos.fr,8765182
christophe11@free.fr,8254343
```

- un module `evaluation.py` où on définit une classe `Evaluation` permettant de créer des échelles d'évaluation attribuant un grade (de `'AA'` à `'ZZ'`) à un score ;

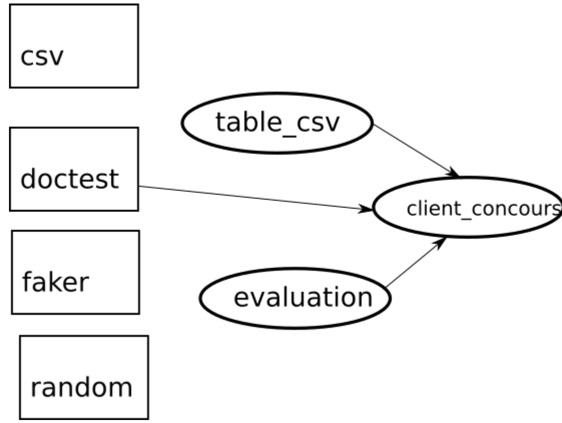
- un module `client_concours.py` où les modules `table_csv.py` et `evaluation.py` sont importés et utilisés par des fonctions clients pour traiter des tables de résultats de concours en attribuant à chaque score un grade :

```
email,score,grade
wmeyer@mercier.fr,6546239,JA
prevostjerome@noos.fr,8765182,NM
christophe11@free.fr,8254343,MI
```

1. Récupérer l'archive `concours_eleves.zip`, l'extraire avec `unzip` puis ouvrir les trois fichiers du projet avec `Spyder`.

2. Dans `client_concours.py` compléter la fonction `generer_csv` en respectant la spécification donnée dans sa `docstring`.

3. Compléter le graphe de dépendances entre modules ci-dessous avec toutes les flèches d'origine un module importé et d'extrémité le module client. Les carrés représentent les modules de la bibliothèque standard (`csv`, `random`, `doctest`) ou d'une bibliothèque installée (faker). Les ellipses entourent les modules du projet qui doivent être tous placés dans le même répertoire.



4. Dans `evaluation.py` est définie une classe `Evaluation`.

Sa méthode `Evaluation.recherche_grade_seq(self, points)` doit associer un grade entre `'AA'`, `'AB'`, . . ., `'ZZ'` à un nombre de points, selon une échelle fixant des seuils pour chaque grade (ordre croissant).

a. Quelques tests sont fournis dans la `docstring`. Ils seront exécutés par la méthode `testmod` du module `doctest` qui est placée dans le bloc principal du programme.

Compléter ce jeu de tests avec d'autres tests pertinents sur les valeurs limites de seuil (nombres de points égal à un seuil, plus grand d'une unité, inférieur d'une unité).

b. Compléter la méthode `Evaluation.recherche_grade_seq` en respectant la spécification donnée dans sa `docstring`.

2 Tester

Objectif 2

Écrire un programme, c'est bien. Écrire un programme juste, c'est mieux. En première on a présenté des outils comme l'invariant de boucle pour démontrer la correction d'un programme c'est-à-dire qu'il produise le résultat souhaité pour une entrée fournie et rien d'autre. Néanmoins il n'est pas toujours possible de prouver la correction d'un programme.
Par ailleurs des erreurs peuvent apparaître lors de l'implémentation d'un algorithme correct.
C'est pourquoi il est intéressant (bien que non suffisant en général) de tester les bouts de code que l'on écrit sur des cas particuliers pour lesquels on connaît bien le résultat attendu.
Citons un extrait du discours du célèbre informaticien Edger Dijkstra prononcé en 1972 lorsqu'il reçut le prix Turing :

> *Mais si tester un programme peut être une technique très efficace pour montrer la présence de bogues, elle est désespérément incapable d'en montrer l'absence. La seule manière efficace de relever significativement le niveau de confiance en un programme est de produire une preuve convaincante de sa justesse.*

2.1 Différents types de tests

Définition 2 ***Vocabulaire des tests***

Source : Document ressource Eduscol « Mise au point de programmes testés » :
https://eduscol.education.fr/document/30058/download

☞ Voici quelques définitions essentielles concernant les tests :

- On appelle **cas de test** un triplet *(descriptif, données d'entrée, résultat attendu)* précisant, pour des données précises, le résultat attendu de la partie du programme que l'on veut tester.
- On appelle **jeu de tests** un ensemble de cas de test destinés à valider une partie précise du fonctionnement d'un programme.

☞ On peut classer les tests selon différents critères :

- le **niveau** du test :
  - **test unitaire** pour tester une petite unité d'un programme (une fonction, une méthode de classe ...);
  - *test d'intégration* pour vérifier que plusieurs unités d'un programme fonctionnent ensemble.
- le **processus de conception** du test :
  - **test « boîte noire »** conçu à partir des données d'entrées potentielles, indépendamment du code écrit. Un test « boîte noire » peut donc être écrit avant le code, ou s'il est écrit après, il doit être écrit par quelqu'un qui ne connaît pas le code;
  - **test « boîte blanche »** conçu à partir du programme. Le but de ce type de test est de couvrir les différents chemins d'exécution : *couverture des instructions, couverture des décisions* (limites de boucles, différentes branches d'instructions conditionnelles), *couverture des conditions multiples* (expressions booléennes complexes).
- le **sujet** du test :
  - un **test fonctionnel** vérifie qu'un programme satisfait bien sa spécification;
  - un *test de performance* évalue la capacité du programme à traiter des entrées de grande taille avec des complexités temporelle et spatiale acceptables;
  - une *test d'utilisabilité* évalue l'ergonomie de l'Interface Homme Machine d'un programme.

Le type de test choisi dépend du niveau de développement d'un programme : on commence par les tests unitaires avant les tests d'intégration voir de performance ou d'utilisabilité. Dans ce cours, on se limite aux tests unitaires mais dans la réalisation d'un projet, une panoplie plus complète pourra être déployée.

2.2 Conception de tests en Python

Méthode

☞ **Méthode 1**

On peut concevoir des tests comme en première à l'aide de vérifications d'assertions définies avec l'instruction `assert` et en regroupant les tests dans un module séparé.

☞ **Méthode 2**

On peut utiliser le module doctest de la bibliothèque standard.

- On écrit les tests dans les chaînes de documentation, selon le format d'une session interactive :

```
>>> expression
resultat
```

- On place à la fin du programme un bloc exécuté uniquement si le module n'est pas importé :

```python
if __name__ == "__main__":
    doctest.testmod(verbose=True)
```

  La méthode `testmod` va évaluer tous les bouts de codes inclus dans les chaînes de documentation du programme, qui correspondent à la syntaxe d'une session interactive et vérifier qu'ils donnent bien le résultat attendu.

L'écriture de tests avec `doctest` est simple mais présente de nombreux inconvénients : les tests ne sont pas séparés du programme, ils surchargent les chaînes de documentation, `doctest` est très sensible à la mise en forme des résultats attendus (un espace en trop et le test échoue!).

Voici un exemple pour une fonction (non optimisée) déterminant si un entier naturel est premier et placée dans un fichier `est_premier.py` :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Exemple de test avec doctest
"""
import doctest

def est_premier(n):
    """
    Détermine si l'entier naturel n est premier

    Parameters: n de type int
    Returns; boolean

    Premier jeu de tests :

    >>> est_premier(0)
    False
    >>> est_premier(1)
    False
    >>> est_premier(2)
    True
    >>> [est_premier(n) for n in range(3, 12)]
    [True, False, True, False, True, False, False, False, True]
    >>> est_premier(57)
    False

    Tests sur les nombre de Fermat :

    >>> [est_premier(2**(2**n) + 1) for n in range(0, 6)]
    [True, True, True, True, True, False]
    """
    # préconditions
    assert isinstance(n, int)
    assert n >= 0
    if n <= 1:
        return False
    diviseur = 2
    while (diviseur <= n) and (n % diviseur != 0):
        diviseur = diviseur + 1
    return diviseur == n

if __name__ == "__main__":
```

```
    doctest.testmod(verbose=True)
```

La sortie s'affiche dans la console de session interactive :

```
Trying:
    est_premier(0)
Expecting:
    False
ok

.........

Trying:
    [est_premier(2**(2**n) + 1) for n in range(0, 6)]
Expecting:
    [True, True, True, True, True, False]
ok
1 items had no tests:
    __main__
1 items passed all tests:
   6 tests in __main__.est_premier
6 tests in 2 items.
6 passed and 0 failed.
Test passed.
```

☞ **Méthode 3** pytest (installation avec `pip install pytest`) est une bibliothèque permettant d'organiser les jeux de tests de façon modulaire :

- Chaque *jeu de tests* inclus dans un fichier dont le nom commence par `test_` est placé dans le même répertoire que le programme à tester. On définit une fonction par *cas de test* en préfixant la fonction par `test_`.
- Depuis une ligne de commandes, on exécute un jeu de tests en mode verbeux avec `pytest -v test_mon_jeu.py`. Un rapport de tests est alors affiché.

Voici un extrait d'un jeu de tests placé dans `test_est_premier.py` pour tester la fonction `est_premier` définie précédemment. On peut capturer les erreurs (ici lors de la vérification des préconditions de type et de valeur positive) plus facilement qu'avec `doctest` :

```
import pytest
from est_premier import est_premier

# cas de tests pour capturer des erreurs prévues

def test_erreur_type_float():
    with pytest.raises(AssertionError):
        est_premier(4.0)

def test_erreur_type_str():
    with pytest.raises(AssertionError):
```

```
        est_premier("4")

def test_erreur_negatif():
    with pytest.raises(AssertionError):
        est_premier(-1)

# autres cas de tests

def test_1():
    resultat = est_premier(1)
    attendu = False
    assert resultat == attendu

def test_2():
    resultat = est_premier(2)
    attendu = True
    assert resultat == attendu

def test_fermat():
    resultat = [est_premier(2 ** (2 ** n) + 1) for n in range(0, 6)]
    attendu = [True, True, True, True, True, False]
    assert resultat == attendu
```

Exemple de rapport pour l'exécution du jeu de tests sans erreurs :

```
(base) administrateur@smob-ubuntu-p01:~$ pytest -v test_est_premier.py
============================== test session starts ===============================
platform linux -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- /opt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /home/administrateur
plugins: anyio-3.5.0, Faker-8.8.1
collected 9 items

test_est_premier.py::test_erreur_type_float PASSED                       [ 11%]
test_est_premier.py::test_erreur_type_str PASSED                         [ 22%]
test_est_premier.py::test_erreur_negatif PASSED                          [ 33%]
test_est_premier.py::test_0 PASSED                                       [ 44%]
test_est_premier.py::test_1 PASSED                                       [ 55%]
test_est_premier.py::test_2 PASSED                                       [ 66%]
test_est_premier.py::test_petits_entiers PASSED                          [ 77%]
test_est_premier.py::test_57 PASSED                                      [ 88%]
test_est_premier.py::test_fermat PASSED                                  [100%]

=============================== 9 passed in 0.08s ================================
```

Exemple de rapport pour l'exécution d'un jeu de tests avec erreurs :

```
(base) administrateur@smob-ubuntu-p01:~$ pytest -v test_est_premier.py
======================================= test session starts ========================================
platform linux -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- /opt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /home/administrateur
plugins: anyio-3.5.0, Faker-8.8.1
collected 9 items

test_est_premier.py::test_erreur_type_float PASSED                                           [ 11%]
test_est_premier.py::test_erreur_type_str PASSED                                             [ 22%]
test_est_premier.py::test_erreur_negatif PASSED                                              [ 33%]
test_est_premier.py::test_0 PASSED                                                           [ 44%]
test_est_premier.py::test_1 PASSED                                                           [ 55%]
test_est_premier.py::test_2 FAILED                                                           [ 66%]
test_est_premier.py::test_petits_entiers PASSED                                              [ 77%]
test_est_premier.py::test_57 PASSED                                                          [ 88%]
test_est_premier.py::test_fermat PASSED                                                      [100%]

============================================ FAILURES ==============================================
____________________________________________ test_2 ________________________________________________

    def test_2():
        resultat = est_premier(2)
        attendu = True
>       assert resultat == attendu
E       assert False == True

test_est_premier.py:40: AssertionError
===================================== short test summary info ======================================
FAILED test_est_premier.py::test_2 - assert False == True
=================================== 1 failed, 8 passed in 0.15s ====================================
```

Exercice 2 ***Année bissextile***

On reprend l'exercice 6 p. 66 du manuel Hachette.

> *Une année posétérieure à* 1582 *(calendrier grégorien) est bissextile si elle est divisible par* 400 *ou pas divisible par* 100 *et divisible par* 4.

Dans un module `bissextile.py`, on veut écrire une fonction `est_bissextile(a)` qui renvoie un booléen indiquant si l'année a est bissextile.

1. Dans un module `test_bissextile.py` écrire un jeu de tests unitaires en « boîte noire » pour la fonction `est_bissextile`.

   Transmettre son jeu de tests à son voisin.

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

....

....

....

2. Coder la fonction `est_bissextile` en séparant la conditionnelle pour chacun des cas.
   Vérifier si la fonction passe les tests pour son jeu de tests et pour celui de son voisin.

   ....

   ....

   ....

   ....

   ....

   ....

   ....

3. Compléter éventuellement son jeu de tests pour que toutes les branches de la conditionnelle soient couvertes (test « boîte blanche »).

   ....

   ....

   ....

   ....

4. Prolongement possible : exercice 15 page 67 corrigé à la fin du manuel Hachette.

Exercice 3 *Test d'une pile*

Faire l'exercice 9 p. 66 corrigé à la fin du manuel Hachette.
On complètera les codes fournis dans l'archive `test_pile.zip`

Exercice 4 *Concours de jeux vidéos partie 2*

On reprend le contexte de l'exercice 1. On dispose d'un fichier `csv` stockant une table de données des résultats du concours de jeu vidéo avec deux descripteurs `'email'` et `'score'`.
On veut évaluer chaque score de la table avec un objet de la classe `Evaluation` définie dans `evaluation.py`.

1. Exécuter `client_concours.py` puis en console interactive, exécuter
   `test_performance_completer_grade()`.

```
>>> test_performance_completer_grade()
Taille=1000 Temps d'évaluation (recherche seq): 0.020000481999886688
    s
```

```
Taille=10000 Temps d'évaluation (recherche seq): 0.170197482999356 s
Taille=100000 Temps d'évaluation (recherche seq): 1.635903872000199 s
Taille=1000000 Temps d'évaluation (recherche seq): 16.2560295459989 s
```

Quelle conjecture peut-on faire sur l'évolution du temps d'évaluation pour des tables de 1000, 10000, 100000, 1000000 de joueurs ?

....

....

....

....

2. Sachant qu'il existe $26 \times 26 = 676$ seuils et grades distincts, estimer le gain de performance que pourrait apporter le remplacement d'une recherche séquentielle par une recherche dichotomique.

....

....

....

....

3. Dans le module `evaluation.py`, importer le module `bisect`.

Consulter son interface avec `help(bisect)` et choisir la fonction qui permettra d'implémenter en quelques lignes la méthode `recherche_grade_dicho(self, points)` de la classe `Evaluation`.

Tester cette méthode avec le jeu de tests `pytest` fourni dans `test_recherche_grade_dicho.py`.

....

....

....

....

....

....

4. Dans `client_concours.py`, compléter `test_performance_completer_grade()` pour afficher également les performances avec une recherche dichotomique. Le gain de performance correspond-il à celui escompté ?

....

....

....

5. Question bonus :

Le module `bisect` est tout petit (80 lignes). Implémenter son interface sans consulter le code source.

Exercice 5 ***TP p. 64 manuel Hachette***

Compléter les codes fournis dans l'archive `tp_test_unitaire.zip`.

3 Mettre au point

Objectif 3

Avant de tester la correction d'un programmation, il faut d'abord écrire un programme dont l'exécution ne « plante ». C'est la phase de ***mise au point*** *du programme.*

3.1 Activité d'introduction

Activité 1 ***Traquer les bugs, activité page 55 du manuel Hachette***

Activité sur Capytale : `https://capytale2.ac-paris.fr/web/c/154b-677638/mcer`.

3.2 Différents types d'erreurs

Définition 3

Pour déboguer un programme, il est important de comprendre les messages d'erreurs affichés par Python. On distingue trois grandes catégories d'erreurs :

- Les **erreurs de syntaxe** sont identifiées par Python lorsqu'il parcourt le code du programme avant de l'exécuter. Elles sont souvent dues à l'oubli de l'indentation, du symbole ' :' pour marquer le début d'un bloc, à l'oubli d'une parenthèse ou crochet fermant.

  Par exemple si Python parcourt le code :

```
for k in range(10)
    print(k)
```

  Il affiche :

```
  File "<input>", line 1
    for k in range(10)
                     ^
SyntaxError: invalid syntax
```

  On peut noter la flèche pointant sur la position de l'erreur de syntaxe.

- Les **erreurs d'exécution** ou **exceptions** sont levées par Python lorsqu'il ne peut pas poursuivre l'exécution du code. Python envoie alors un message sur la sortie d'erreur. On parle de *traceback* car le message contient la pile des appels de fonctions en cours au moment où l'exception est levée. De

haut en bas (`most recent call last`), on parcourt la pile de son sommet (dernière fonction appelée) jusqu'à sa base (première fonction appelée).

Par exemple si on exécute le code suivant :

```
def appel1(n):
    return n / 0

def appel2(n):
    return appel1(n)

def appel3(n):
    return appel2(n)

appel3(1)
```

Le message de *traceback* suivant est affiché :

```
Traceback (most recent call last):
  File "<input>", line 11, in <module>
  File "<input>", line 9, in appel3
  File "<input>", line 6, in appel2
  File "<input>", line 3, in appel1
ZeroDivisionError: division by zero
```

Les **principales exceptions** sont :

Exception	Contexte
`NameError`	accès à une variable globale inexistante
`UnboundLocalError`	accès à une variable locale inexistante
`IndexError`	accès à un indice invalide d'un tableau
`KeyError`	accès à une clef inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	opération appliquée à des valeurs de types incompatibles
`RecursionError`	dépassement de la taille maximale de la pile d'appels récursifs

☞ Les **erreurs logiques** ne lèvent pas d'exception mais elles correspondent à des situations où le programme ne fournit pas le résultat attendu. Elles peuvent être mises en évidence par les tests déjà étudiés.

Exercice 6 ***Identifier les principales exceptions***

Faire l'exercice dans le fichier Capytale `https://capytale2.ac-paris.fr/web/c/25cf-629079/mcer`

3.3 Localiser une erreur

Méthode

Pour localiser une erreur, d'exécution ou une erreur logique, on peut utiliser les méthodes suivantes :

☞ **Méthode 1 : exécution pas à pas à la main** On exécute pas à pas le code en traçant l'avancement des variables dans un tableau.

- **Méthode 2 : insérer des affichages avec des `print`** On peut ainsi contrôler plus finement une exécution pas à pas, en focalisant uniquement sur les pas d'exécution identifiés comme sensibles (valeur d'un variant ou d'un invariant en entrée/sortie de boucle, passage dans une branche de conditionnelle …)

- **Méthode 3 : utilisation d'un débogueur** Un *débogueur* est un outil logiciel fourni par un environnement de programmation qui permet d'automatiser les méthodes précédentes sans modifier le code avec l'insertion de `print`.

*Les jeux de tests « boîte noire » prédéfinis jouent un rôle important dans la validation d'un correctif d'erreur. On garantit ainsi la **non régression**.*

Exercice 7 ***Partage de tableau, exo 16 p. 67 du manuel Hachette.***

Extraire l'archive `partage_tableau.zip` qui contient les modules :

- `partage_tableau.py` où la fonction `partage(t)` sera écrite.
- `test_partage_tableau.py` où on placera les tests unitaires `pytest` de la fonction `partage(t)`.

Pour la question 2., on pourra s'aider du débogueur de `Spyder` et de l'explorateur de variables pour localiser les erreurs. On placera un *point d'arrêt* juste après l'entrée de boucle.

Documentation du débogueur : https://docs.spyder-ide.org/current/panes/debugging.html.

3.4 Prévenir les erreurs avec de bonnes pratiques

Méthode ***Voir manuel Hachette page 58 à 61***

Adopter de bonnes pratiques dans l'écriture du code permet d'éviter des erreurs.
Une référence : https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/.

- **Style :**

  - Réfléchir avant de coder : ne pas hésiter à travailler au brouillon avec papier et crayon !
  - Utiliser des noms de variables explicites facilite le débogage et la relecture du code.
  - Identifier les unités fonctionnelles et les structures de données : découper son code en fonctions et en classes. Écrire des petites fonctions (pas plus 30 lignes) et tant qu'il le faut redécouper plus finement. Si le projet est important, le décomposer en modules.
    On parle de **factorisation de code**.
  - Limiter les imbrications de structures à une profondeur maximale de 3, factoriser les boucles internes par des fonctions.
  - Simplifier les expressions booléennes complexes : parenthéser les opérandes pour réduire les ambiguités, factoriser les opérandes par des variables ou des fonctions.

- **Spécifier, commenter :**

  - Commenter les parties les moins évidents du code et uniquement celles-ci.

- Spécifier une fonction avant d'écrire son code (`docstring` + préconditions) et s'assurer que la spécification est bien satisfaite lorsqu'on l'appelle : ordre et type des paramètres et des valeurs retournées (vérifier qu'il y a un `return` et qu'il est bien placé).
- Expliquer son code à un camarade.

☞ **Tester, déboguer :**

- Écrire des jeux de tests unitaires « boîte noire » avant de coder les fonctions les moins évidentes. Vérifier les tests après chaque modification du code.
- Lire attentivement les messages d'erreur, exécuter le programme en pas à pas pour localiser une erreur.

☞ **Terminaison et correction :**

- S'assurer que la condition d'entrée dans une boucle `while` devient fausse, vérifier le sens des inégalités dans une condition de boucle.
- Vérifier les bornes dans une boucle `for`, attention la borne supérieure de `range` est exclue.
- Insérer des assertions dans le code pour vérifier *variant* et *invariant* de boucle.
- S'assurer qu'une fonction récursive se termine ⇒ les appels doivent converger vers un cas de base.

☞ **Complexité :** L'ordinateur est fini, ses capacités de calcul sont limitées dans le temps et l'espace :

- *Temps :* Mesurer la performance de son code sur des entrées de grande taille, changer d'algorithme ou de structure de données si la complexité temporelle est mauvaise.

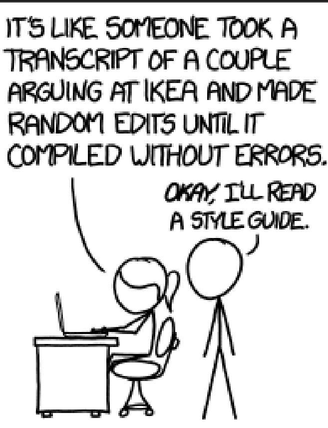
Complexités à connaître, par ordre croissant

Complexité	Evolution de la taille de l'entrée n	Evolution du temps d'exécution
Constante	$\times 2$	constant
Logarithmique	$\times 2$	ajout d'une constante
Linéraire	$\times 2$	$\times 2$
Quadratique	$\times 2$	$\times 2^2$
Cubique	$\times 2$	$\times 2^3$
Exponentielle	$\times 2$	au carré

- *Espace :* Changer de structure de données en cas de dépassement de capacité mémoire. Garder à l'esprit que l'ordinateur ne peut représenter de façon exacte qu'un échantillon fini de nombres :
  - Type `int` : tous les entiers relatifs mais uniquement entre certaines bornes (cela dépend des langages, en Python la taille de la mémoire fixe la limite).
  - Type `float` : uniquement certains décimaux avec un développement fini en base 2 de la forme $\frac{p}{2^n}$ ⇒ un test d'égalité sur les flottants n'a pas de sens, il faut le remplacer par une mesure de proximité acceptable par exemple avec la fonction `math.isclose`.

Exercice 8 *Factorisation de code*

Traiter l'exercice dans le fichier Capytale https://capytale2.ac-paris.fr/web/c/1654-629029/mcer.



Source : *XKCD Code Quality*

Table des matières

1 Modularité **1**

2 Tester **5**
- 2.1 Différents types de tests 5
- 2.2 Conception de tests en Python 6

3 Mettre au point **13**
- 3.1 Activité d'introduction 13
- 3.2 Différents types d'erreurs 13
- 3.3 Localiser une erreur 14
- 3.4 Prévenir les erreurs avec de bonnes pratiques 15