BONNES PRATIQUES

À la fin de ce chapitre, je sais :

- ☞ produire une code intelligible
- ☞ structurer un programme
- ☞ tester et se laisser guider par les tests
- ☞ utiliser la programmation défensive

La notion de bonnes pratiques en informatique peut paraître naïve et inutile à un débutant : le code marche ou ne marche pas selon une spécification donnée. En fait, la réalité est plus nuancée. Deux méthodes (parfois complémentaires) sont actuellement utilisées pour garantir le bon fonctionnement d'un logiciel :

1. le test exhaustif du code,
2. la vérification formelle (c'est-à-dire la preuve mathématique de la correction) du code.

D'un côté, l'utilisation des langages courants de développement (C, C++, Java, Javascript, Python, Scala…) rend les erreurs inévitables. D'un autre côté, les méthodes formelles ont à la fois des limites et induisent un coût et une lourdeur non négligeable sur le développement.

Les bonnes pratiques sont donc celles qui permettent à la fois de réduire les erreurs logicielles et leurs conséquences tout en limitant l'impact sur le coût de développement. À l'heure des cyberguerres, ces pratiques ont un impact conséquent et sont vitales au développement dans le domaine informatique.

A Défendre le code : pourquoi et comment ?

Il est possible d'adopter un style de programmation **défensif**, style qui renforce la fiabilité, la sécurité et rend les erreurs plus faciles à comprendre.

a Différents types d'erreurs

Différentes erreurs peuvent être distinguées :

- des erreurs dans la gestion des structures, des types et des conteneurs : types des variables, bornes d'un tableau ou d'une liste, référence mémoire erronée, boucle qui ne

termine jamais... Ces erreurs sont de la responsabilité unique du développeur et il peut donc les corriger s'il les perçoit !

- des erreurs qui dépendent de l'environnement : lecture/écriture d'un fichier, d'un socket réseau, d'un capteur, d'un clavier, d'un écran, d'un périphérique quelconque... Ces erreurs ne sont pas inhérentes au programme : elles sont parfois indétectables car dépendantes de l'environnement. Cependant, il faut absolument les envisager afin qu'elles ne soit pas une cause d'échec du programme. Les exceptions ou les promesses (promises Javascript ou future Java et Rust) sont des bonnes solutions pour gérer ce type d'erreurs.
- des erreurs (souvent fatales) en lien avec les processus de la machine elle-même : dépassement de la taille de la pile d'exécution, dépassement de capacité dans le cadre de la représentation des nombres, limites du langage, du compilateur ou de l'interpréteur, du processeur ou du système d'exploitation. Ces erreurs remettent souvent cause le programme et ses fondements. Si modifier le programme n'est pas possible, elles nécessitent alors un ajustement de la plateforme.

b Des exceptions plutôt que des codes retours

■ **Définition 1 — Exception.** Une exception est une interruption de l'exécution d'un programme à cause d'une erreur dont la possibilité a été **prévue**.

R À l'origine de l'anticipation de cette erreur, on peut trouver soit le programmeur soit le langage utilisé. Certains langages sont plus prévoyants que d'autres !

Lorsqu'une exception est levée par une partie du code, une routine d'interruption est exécutée et le programme est interrompu : celui-ci s'arrête là et n'achève pas son exécution. Cependant, si le programmeur anticipe la levée de cette exception, il peut la capturer, la gérer et continuer l'exécution de son programme (cf. exemples 1 et 2).

■ **Exemple 1 — Division par zéro.** Imaginons qu'un processus d'entrée/sortie [a] a initialisé une variable `b` à zéro. Imaginons que, dans la suite du code, le calcul de `a/b` soit nécessaire. Alors, on peut anticiper le problème en utilisant une exception comme suit :

```
try:
    c = a/b
except ZeroDivisionError as e:
    print("Error: Cannot divide by zero")
    # faire quelque chose d'autre ou rien si ce n'est pas critique
# ici le programme continue même lorsque l'exception a été levée
```

a. un capteur par exemple

■ **Exemple 2 — Lecture d'un fichier qui n'existe pas.** L'exemple le plus classique d'exception est celui engendré par la lecture d'un fichier qui n'existe pas lors de l'exécution du

programme. Le développeur ne peut pas savoir si le fichier existe, par contre il peut anticiper le cas où celui-ci n'existe pas, comme c'est le cas ci-dessous, en créant le fichier par exemple.

```
import logging
try:
    with open("file.log") as file:
        read_data = file.read()
except FileNotFoundError as e:
    print(e)
    logging.basicConfig(filename="file.log", level=logging.INFO)
# le programme peut continuer nominalement
```

R Pour signifier à l'utilisateur d'une fonction que l'exécution de celle-ci s'est déroulée sans erreur, les programmeurs ont longtemps utilisé [a] la technique du code retour. Par convention, si la fonction renvoyait :

- 0, alors elle s'était déroulée sans erreur,
- un nombre négatif, alors elle s'était déroulée avec erreur et le nombre négatif indique le type de l'erreur.

Cependant, cette pratique nécessitait de tester systématiquement le code renvoyé par la fonction. On préfère aujourd'hui utiliser des exceptions pour signifier un dysfonctionnement de la fonction. Les paramètres renvoyés sont les sorties et l'exception peut être standardisée ou personnalisée.

a. et utilise toujours en langage C

c Des assertions qui lèvent des exceptions

■ **Définition 2 — assertion.** Une assertion est une expression booléenne, c'est-à-dire un test évalué à vrai ou faux. Si l'évaluation échoue, une exception est levée signifiant ainsi l'échec du test.

En Python, c'est le mot-clef `assert` qui permet d'écrire des assertions comme indiqué ci-dessous.

```
number = 42
assert number > 0
assert number > 0, f"expect number greater than 0, but got: {number}"
assert number > 0 and isinstance(number, int), f"natural number expected, got: {number}"
```

B Spécifier

Bien programmer commence par bien spécifier ce que l'on attend du logiciel.

■ **Définition 3 — Spécification.** Une spécification est un ensemble explicite d'exigences qu'un produit à concevoir doit satisfaire.

Les exigences de spécification servent de base à la conception d'un produit et à la validation du produit réalisé.

La spécification fonctionnelle permet de définir les différentes fonctions que le logiciel réalise. Elle se décline selon trois axes :

1. décomposition : décomposer la fonction en sous-fonctions,
2. données : quelle est la nature des données d'entrée et de sortie (unités, types, ordre de grandeur)
3. comportement : comment se comporte la fonction selon le scenario (que se passe-t-il par exemple lorsque l'utilisateur fournit des données d'un mauvais type ou en dehors d'un intervalle donné ?)

À cette étape, on considère le logiciel comme une boite noire et on s'intéresse aux entrées et aux sorties des fonctions. C'est pourquoi les annotations de types sont très utiles en pratique : la signature spécifie une fonction, sans préjuger de l'implémentation.

■ **Exemple 3 — Annotation de type et spécification fonctionnelle.** En Python, comme dans d'autres langages, la signature d'une fonction peut être très précise grâce à l'annotation des types. Par exemple, une fonction de signature `sort(L : list)—> None` peut être considérée comme une boîte noire dont l'entrée est constituée d'une liste que l'on peut modifier et qui ne renvoie rien.

Les préconditions et les postconditions sur les données se traduisent généralement par des assertions qui portent sur les paramètres.

■ **Exemple 4 — Précondition et postcondition.** Lorsqu'une fonction a des exigences précises sur des données d'entrée ou de sortie, on peut utiliser des assertions pour vérifier en amont et en aval que ce qui est donné en entrée et ce qui sort de la fonction est correct.

```
def temp_convert(tk: float) -> float:
    # précondition : la température en Kelvin est toujours positive
    assert tk >= 0, "Tempature in Kelvin should be positive."
    return tk - 273.15
```

La spécification peut également préciser qu'un certain type d'exception est levée si la température d'entrée est négative. L'utilisation d'exceptions dédiées est possible dans de nombreux langages. Elle améliore le développement en facilitant l'analyse des erreurs.

```
class InvalidKelvinTemperatureException(Exception):
    """Exception raised for errors in the input temperature.

    Attributes:
        temperature -- input temperature which caused the error
        message -- explanation of the error
    """
```

```
        def __init__(self, temp, message="Temperature should be positive if in Kelvin"):
            self.temp = temp
            self.message = message
            super().__init__(self.message)


    def temp_convert(tk: float) -> float:
        if tk < 0:
            raise InvalidKelvinTemperatureException(tk)
        return tk - 273.15



    # TESTS
    try:
        temp_convert(-35.5)
    except Exception as e:
        print(f"Expected exception --> {e}")
    print(temp_convert(45.2))
```

Dans le flot de conception d'un produit, de nombreuses étapes suivent la spécification fonctionnelle. Parmi celles-ci, on peut citer notamment la spécification logique (composants abstraits : services, clients et leurs interactions) et la spécification physique ou matérielle (composants concrets : machines, ressources, réseaux, mémoires).

C Écrire un code intelligible

Bien programmer nécessite de produire un code intelligible.

a Nommer des variables

■ **Définition 4 — Conventions.** Règles de conduite adoptées à l'intérieur d'un groupe social. Exemple : en mathématiques, l'inconnue c'est x.

D'une manière générale, en informatique contemporaine, on préfère les conventions aux configurations [1], car cela fait gagner un temps précieux. Respecter des conventions dans l'écriture des programmes permet leur traitement automatique par d'autres outils pour générer d'autres programmes (tests, vérification, documentation, mise en ligne, interfaçage). Cela permet également aux développeurs de comprendre plus rapidement un code.

L'intelligibilité est certainement la plus grande qualité qu'on puisse exiger d'un code. Aujourd'hui, dans la plupart des entreprises, des normes de codage sont appliquées afin de rendre le travail collaboratif plus efficace. Pour le langage Python, un ensemble de recommandations a été produit (cf. Python Enhancement Proposals - PEP 8). Donc, si vous ne savez pas trop comment faire, il est toujours possible de s'y référer. Les IDE actuels implémentent ces recom-

1. c'est à dire des arrangements spécifiques.

mandations et peuvent formater ou proposer des changements conformes aux Python Enhancement Proposals.

Méthode 1 — Choisir un nom de variable ou de fonction *Mal nommer les choses c'est ajouter au malheur de ce monde*[a]. Alors tâchons de bien les nommer.

À faire :

- choisir des noms de variables et de fonctions en minuscules.
- si plusieurs mots sont nécessaires, mettre un _ entre les mots,
- préférer un **verbe** pour les fonctions, par exemple `trier`
- préférer des **noms** pour les variables, par exemple `color`)
- appeler un chat un chat.

À ne pas faire :

- utiliser les lettres l,O ou I pour un nom de variables. Elles sont confondues avec 1 ou 0.
- utiliser autre chose que les caractères ASCII (par exemple des lettres accentuées, des lettred grecques ou des kanjis),
- appeler une variable d'après un mot clef du langage Python (lambda, list, dict, set, global, try, True…)

a. citation apparemment de Brice Parrain dans une réflexion sur l'étranger d'Albert Camus. À vérifier cependant.

■ **Exemple 5 — Noms courants de variables.** Voici des noms possibles et courants pour :

- les types `int` : `i,j,k,m,n,p,q,a,b`
- les types `float` : `x,y,z,u,v,w`
- les accumulateurs : `acc, s, somme, prod, produit, product`
- les pas[a] : `dt, dx, dy, step, pas`,
- les chaînes de caractères : `c, ch, s`,
- les listes : `L, results, values`,
- les dictionnaires : `d, t, ht`,
- les constantes en majuscules : `MAX_SIZE`.

a. c'est à dire la distance entre chaque élément d'un vecteur temporel par exemple

b Commenter un code

Méthode 2 — Faire un commentaire Un commentaire peut être utile dans une copie afin de :

- détailler un point du code calculatoire,
- expliquer un choix d'implémentation de structure,
- mettre en exergue un variant (terminaison) ou un invariant (correction) du code,
- mettre en avant une pré-condition ou une post-condition.

Cependant, d'une manière générale, il n'est pas forcément nécessaire, pourvu que votre code soit intelligible. Le choix des noms des variables, le respect de l'indentation et des conventions sont les clefs d'un code intelligible.

Que peut-il arriver de pire à quelqu'un qui raconte une blague? Devoir l'expliquer. Il en est de même du commentaire. Le commentaire peut être utile à l'intelligibilité mais il peut lui nuire également. Par ailleurs, dans le temps, un commentaire peut faire référence à une ligne modifiée ou une variable dont le nom a changé. Dans ce cas là, le commentaire nuit à la compréhension.

On s'attachera donc à :

- n'inscrire que des commentaires utiles et brefs,
- s'assurer qu'ils sont cohérents avec le code,
- à ne pas paraphraser le code.

On préfèrera écrire un code directement lisible. Sur le site de ce cours, vous trouverez des exemples de copies de concours que je vous aide à analyser. Bien écrire et bien nommer permet de gagner des points.

c Segmenter le code en modules réutilisables

La programmation structurée permet de décomposer un code en sous-parties, les fonctions et les bibliothèques de fonctions. On s'attachera donc à créer des codes construit à partir de fonctions réutilisable afin de ne pas réinventer la poudre. Celles-ci trouveront leur place dans bibliothèques de fonctions (des modules Python) et pourront être réutilisée à volonté par vous et les personnes avec qui vous souhaiterez travailler.

D Tester son code et le maintenir

Pour être un minimum fiable, un code doit être testé. Un logiciel doit également tenir dans le temps, durer et évoluer en fonction des besoins. Cette maintenance logicielle est cruciale et très importante : un code est rarement figé, il est régulièrement mise à jour. Comment s'assurer alors que, malgré l'évolution, ce qui fonctionnaire avant, fonctionne toujours?

Un bon code est donc un code bien testé ou, encore mieux, un code développé à partir des tests (test driven).

a Types de tests

■ **Définition 5 — Tests statiques.** Les tests statiques sont opérés en dehors de l'exécution

du programme. Il s'agit de relectures du code par un humain ou par un logiciel dans le but de détecter des anomalies.

■ **Définition 6 — Tests dynamiques.** Les test dynamiques sont opérés lors de l'exécution du code.

On distingue également :

1. les tests unitaires qui portent sur les fonctions élémentaires du code,
2. les tests d'intégration qui portent sur les interfaces entre les sous-systèmes. Ce sont des tests comportementaux qui vérifient si les interactions entre les composants logiciels sont nominales.
3. les tests systèmes portent sur l'ensemble du logiciel.
4. les tests de non-régression qui vérifient si l'évolution du code n'a pas créé des bogues ni endommagé les fonctionnalités pré-existantes à l'évolution (qui fonctionnaient avant !),
5. les tests de robustesse et de performance qui viennent stresser le système et le pousser dans ses retranchements.

b Données de test

Pour créer des tests, il est fondamental de créer les données d'entrée des tests, conformément aux spécifications. Dans le cadre des tests unitaires, il s'agit de tester des données d'entrées nominales tout comme des données aberrantes. Dans le cadre des tests d'intégration, on est parfois amené à simuler un des sous-systèmes pour fournir des entrées ou des sorties.

À la fin du test, les données recueillies sont dépouillées afin de valider ou non le test.

c Couverture du code

Lors des tests, il est important de s'assurer que toutes les lignes du code ont été testées. En effet, lorsque plusieurs chemins d'exécution sont possibles, si les tests ne sont pas correctement rédigés, il est possible que certaines parties du code ne soient jamais exécutées et donc potentiellement boguées.

Dans le cas où les tests sont correctement rédigés et qu'il existe à la fin de ceux-ci des chemins d'exécution qui n'ont pas été parcourus, alors il est fort probable que cette partie du code ne serve à rien !

d Tests, IDE et bibliothèques

Tous les langages possèdent des bibliothèques qui facilitent l'écriture de tests, tout particulièrement les tests unitaires. Python propose par exemple la bibliothèque `unittest`. Un exemple d'utilisation est donné ci-dessous.

```
import random
import unittest

def euclid(a, b):
```

```
        r = a
        q = 0
        while r >= b:
            r_prec = r
            r = r - b
            q = q + 1
        return q, r

    class TestSum(unittest.TestCase):
        def test_int(self):
            """
            Test that it works on random integers with
            """
            for _ in range(100):
                a = random.randrange(100)
                b = random.randrange(99)
                q, r = euclid(a, b)
                self.assertEqual(q, a // b)
                self.assertEqual(q, a // b)

        # Toutes les méthodes commençant par test_ seront exécutées dans le main

    if __name__ == '__main__':
        unittest.main()
```