CM 1 : Habitudes d'écriture & Tests unitaires

Info1.Algo1

2022-2023 Semestre Impair

Plan

1. Habitudes d'écriture
   - Introduction
   - Nommer
   - Commenter
   - Factoriser
   - Décomposer
2. Tests unitaires
   - Introduction
   - Jeux de tests
   - Mise en oeuvre
   - Compléments

Introduction

Objectifs

- Sensibiliser à quelques dispositifs qui permettent d'améliorer la qualité d'un code (*Clean code*) afin qu'il réponde mieux aux attentes de ceux à qui il s'adresse (relecteurs, collaborateurs, clients, ...)
- Sensibiliser à la nécessité professionnelle de "bien écrire"

L'objectif principal du *Clean code* est la construction de logiciels en minimisant le coût du changement (évolution du besoin, changement de l'équipe qui gère, ...) et en maximisant la capacité à répondre au changement.

Introduction

Pistes de travail

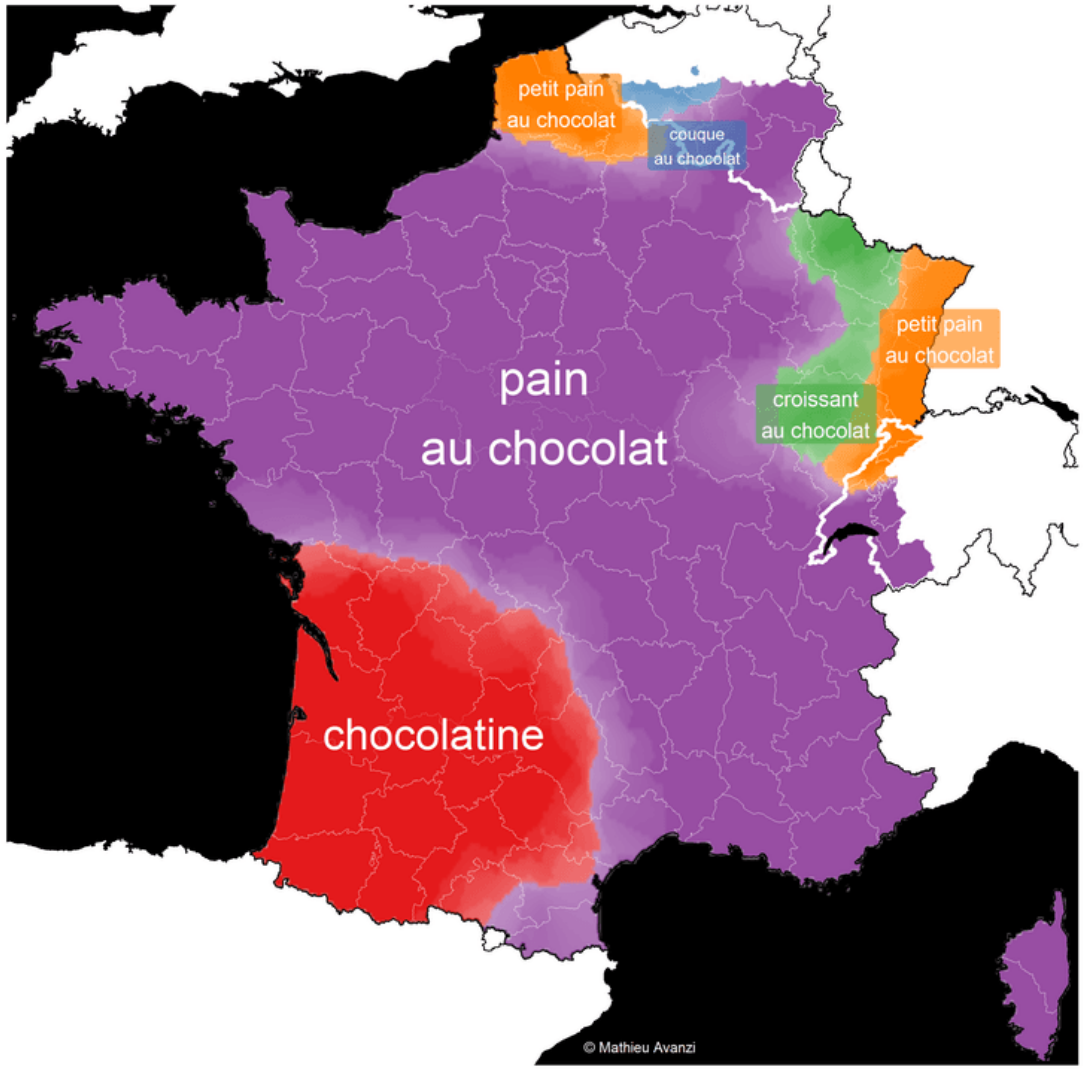
- **Nommer** les variables et fonctions selon des conventions propres au contexte de travail.
- **Commenter** le code pour l'expliquer ou le structurer.
- **Factoriser** les fonctionnalités redondantes.
- **Décomposer** en fonctions, fonctions auxiliaires et code principal.

Autres pistes

- **Keep It Simple, Stupid (KISS)**, éviter l'**optimisation prématurée**.
- Règle du **boy scout** : laisser le code source plus propre que quand on l'a trouvé.

Nommer

Si fueris Romae, Romano vivito more; si fueris alibi, vivito sicut ibi.
Ambroise de Milan?
(IVe siècle)
À Rome fais comme les Romains!



Nommer

Principe général

- Les conventions **peuvent changer** selon le langage, l'entreprise, les personnes avec qui on travaille.
- Compétence visée : **s'adapter** et les utiliser selon le contexte.

Ce qui suit décrit les conventions que nous essaierons d'utiliser dans l'**UE Info1.Algo1**.

Nommer

Conventions pour l'UE Info1.Algo1

- En **français** (pour la bonne compréhension).
- Utiliser des **noms précis**.
- Nom de **variable** : reflète ce que représente la variable.
- Nom de **fonction** :
  - reflète ce que fait une fonction (**le quoi**, pas le comment)
  - si possible avec un **verbe d'action**.
- Refléter le **niveau d'abstraction** auquel on travaille *(nom fonctionnel au niveau fonctionnel, un nom technique au niveau technique, ...)*

Nommer

Plus précisément :

Conventions pour l'UE Info1.Algo1

- Préconisations du langage Python : `snake_case`
  (Voir PEP 8 – Style Guide for Python Code)
- Variables et fonctions **booléennes** : commencent par `est_...` ou `a_...`
- Distinguer une `valeur` et son **indice** `i_valeur` dans un tableau ou une liste.
- **Listes** : `liste_valeurs` ou `valeurs` (le pluriel peut suffire).
- Variables de comptage : `nb_...`

Commenter

Pour expliquer

- En cours de développement : indiquer **l'intention** pour les lignes suivantes *(quitte à supprimer ensuite)*.
- Certains (tous les?) commentaires peuvent être évités quand le nommage est bien fait.
- Indiquer **le pourquoi** et **pas le comment**.

Commenter

Pour structurer

Sur un **code long** *(exemple : > 1000 lignes)*, utiliser les commentaires pour **identifier la structure du code** :

- **Titres et séparations** entre les parties fonctionnelles.
- Parties de **petite taille**.
- Ne pas dépasser les **80 caractères par ligne**.

Factoriser

Principe général

Pour **factoriser** du code :

- Repérer des fonctions dont le code est très ressemblant.
- **Mettre en commun** (factoriser) les parties de code identique.
- Les différences amènent à rajouter **un ou plusieurs paramètres(s) supplémentaire(s)**.

Attention

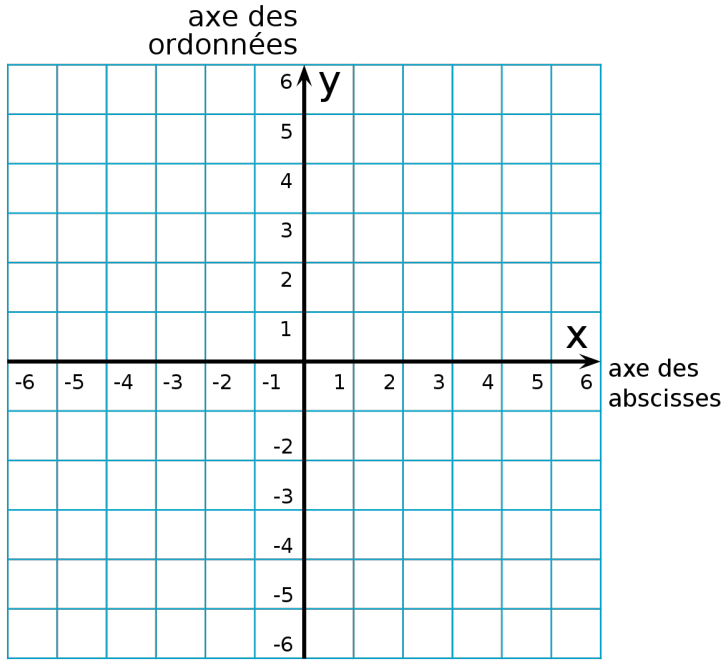
Copier-coller du code $\rightarrow$ produire du code presque identique $\rightarrow$ code factorisable

Factoriser

Exemple

On considère un point mobile repéré par ses coordonnées `(x,y)` dans repère. **Factoriser** les trois fonctions `monter`, `descendre` et `droite` qui permettent de déplacer ce point mobile.

```
1  def monter(point):
2      x,y = point
3      return (x,y+1)
4
5  def descendre(point):
6      x,y = point
7      return (x,y-1)
8
9  def droite(point):
10     x,y = point
11     return (x+1,y)
```



Décomposer

On commence par le **code principal** :

```
1 # CODE PRINCIPAL
2 entrees = lecture_des_donnees()
3 print(fonction_de_traitement(entrees))
```

On remonte en précisant les **étapes du traitement** :

```
1 def fonction_de_traitement(entrees):
2   fonction_qui_fait_ceci(...)
3   fonction_qui_fait_cela(...)
4   return ...
```

Décomposer

Ce qui donne :

```
1  # FONCTIONS UTILITAIRES
2  def fonction_qui_fait_ceci (...) :
3      ...
4
5  def fonction_qui_fait_cela (...) :
6      ...
7
8  # TRAITEMENT PRINCIPAL
9  def fonction_de_traitement(entrees):
10   fonction_qui_fait_ceci (...)
11   fonction_qui_fait_cela (...)
12   return   ...
13
14 # CODE PRINCIPAL
15 entrees = lecture_des_donnees()
16 print(fonction_de_traitement(entrees))
```

Décomposer

Exemple : brouillard de guerre

Le code suivant (correct) permet à l'utilisateur de :

- Entrer une liste de paires `(nom_objet,coordonnees)`.
  Exemple : `[('tresor',(3,-1)),('hutte',(-2,-2)),...,('arme',(4,4))]`
- Entrer les coordonnées `(x,y)` du joueur.
- Afficher les objets à moins de 3 unités du joueur.

```
1 l1 = eval(input())
2 l2 = []
3 p = eval(input())
4 for i in range(len(l1)) :
5     if ((l1[i][1][0]-p[0])**2 + (l1[i][1][1]-p[1])**2)**0.5 < 3.0 :
6         l2.append(l1[i][0])
7 print(l2)
```

Décomposer ce code en fonctions.

Plan

1. Habitudes d'écriture
   - Introduction
   - Nommer
   - Commenter
   - Factoriser
   - Décomposer
2. Tests unitaires
   - Introduction
   - Jeux de tests
   - Mise en oeuvre
   - Compléments

Introduction

Cadre du problème

Le cadre d'étude proposé est celui des **fonctions**. Il est cependant **généralisable** à tout code présentant une ou plusieurs entrées, et une ou plusieurs sorties (programme principal, portion de code).

On exclut les situations suivantes :

- **Interactions** avec l'utilisateur : les entrées sont données simultanément, les sorties sont recueillies simultanément.
- **Effets de bords** non-testables (affichage, ...)

Objectif

Étant donnée une fonction, on souhaite répondre à la question suivante :

La fonction effectue-t-elle bien ce qu'on attend qu'elle fasse?

Jeux de tests

Vocabulaire

Un **jeu de test** permet de **spécifier**, pour une fonction et une **entrée** donnés, la **sortie attendue**.

Méthode

Afin de vérifier si une fonction répond au problème posé :

- Tester cette fonction sur des **jeux de tests** couvrant le **maximum de cas possibles**.
- Trouver les cas **particuliers**, les cas **extrêmes**, les cas **généraux**.

Objectif : mettre en défaut le code à tester.

On étudie la couverture de tests, les domaines des entrées, des sorties. Cette étude se poursuivra lors de l'étude de la **spécification de fonction**.

Jeux de tests

Exemple

On considère la fonction `est_palindrome` qui accepte en paramètre une liste d'entiers et retourne le booléen indiquant si cette liste est (`True`) ou non (`False`) un palindrome.
Écrire des jeux de tests pour cette fonction.

Mise en oeuvre

Principes pour les tests

- **Éviter les interactions utilisateur** (saisie, affichage) exigeant une vérification humaine entre **l'obtenu** et **l'attendu**.
- **Automatiser les tests**.
- **Regrouper les tests** dans une fonction que l'on peut décider d'appeler ou non.

Mise en oeuvre

Définition

Une **assertion** est une **expression booléenne** qui doit être évaluée à **vrai**.
Si l'évaluation de l'assertion est évaluée à **faux**, l'exécution du programme **s'arrête**.

(fonctionnalité présente dans quasiment tous les langages)

L'instruction assert

Syntaxe :

```
1 assert <expression booleenne>, "message d'erreur"
```

Sémantique : Le message d'erreur est affiché si l'assertion est évaluée à faux, sinon l'exécution est silencieuse

Mise en oeuvre

Fonction de test

On regroupe tous les jeux de tests associés à une fonction donnée dans une **fonction de test**.

Nommage :

```
1 def test_<nom fonction a tester>(...):
2   ...
```

Exemple (suite)

Écrire la fonction de test de la fonction `est_palindrome`.

Compléments

À lire : https://realpython.com/python-assert-statement/

Assertions et efficacité

- Les instructions `assert` peuvent être **coûteuses en temps d'exécution ou en mémoire**.
- Après la phase de mise au point, désactiver toutes les instructions `assert` en lançant python avec l'option `-O` ou `-OO` (mode optimisé) :

```
1  python3 -O fichier.py
```

Compléments

Attention!

Instruction `assert` : **exclusivement réservé au débogage**.

Les instructions `assert` sont automatiquement désactivées lorsque l'on lance python en mode optimisé. Elles ne doivent donc pas intervenir dans le fonctionnement d'un programme autrement que pour en vérifier la **correction de l'écriture**.

En particulier :

- Les expressions booléennes utilisées dans les assertions **ne doivent pas avoir d'effets de bord**.
- Les assertions ne doivent être utilisées **ni pour valider des entrées utilisateurs, ni pour valider le contenu de fichiers**.

Compléments

L'instruction `assert` permet de vérifier une assertion **exécutable**. Elle ne doit pas être confondue avec la vérification d'assertions formelles (mot-clef `assert`) qui permettent de guider un outil de preuve d'algorithme.

Compléments

Pour aller plus loin...

Utiliser une librairie dédiée (`unittest`, ...) pour :

- structurer les tests (en modules, classes et fonctions)
- disposer d'un jeu d'assertions plus riche
- lancer automatiquement les tests
- obtenir un feedback d'exécution de tests pour chaque test

Vu dans l'UE ILU1.