UNIVERSITE TOULOUSE III PAUL SABATIER
Faculté des Sciences et d'Ingénierie

Méthodologie

Intro et Bonnes Habitudes : CTD 1

Algorithmique

Semestre 2 - 2019-2020

UE d'Algo - Organisation

Fonctionnement

- 2h de CTD par semaine
- 2h de TP par semaine sur la plateforme PIXAL
- Toutes les informations passeront sur Moodle : Suivi obligatoire
- Evaluation
  - CCTP (TP et DM) 10%. La présence en TP est obligatoire.
  - CR (2h) au milieu du semestre 20%
    - En amphi, QCM, réponses rédigées courtes, algorithmes répondant à des contraintes
  - CT (2h) en fin de semestre 70%
    - Pixal (avec éventuelle correction complémentaire par enseignant)
    - Eventuel QCM

Motivations et Objectifs du Semestre

Revoir la conférence du S1 : "The Big Bug Theory" de J-B. Raclet
`https://moodle.univ-tlse3.fr/pluginfile.php/221211/mod_resource/content/11/main.pdf`

- *Connaître le langage Python n'est pas un but mais un moyen*

Planning prévisionnel

- Bonnes habitudes et Tests (couverture et conception)
- Introduction aux Structures de Données Standard et leurs opérations (Tableau/Matrices)
- Initiation aux Invariants informels
- Base de la récursivité
- SdD Listes et Arbres
- Récursivité avancée et Tris
- SdD Dictionnaires et Set
- SdD Enregistrement, piles, files
- Efficacité / complexité

Motivations et Objectifs de la séance

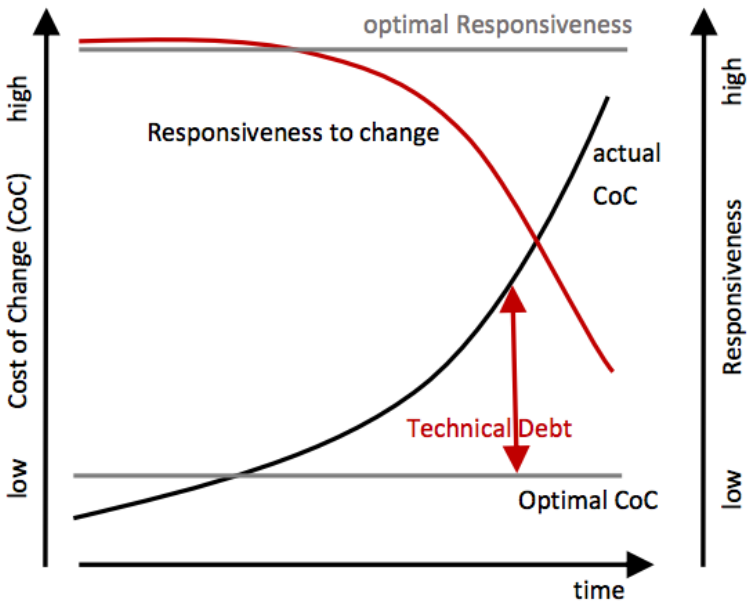
- Installer de bonnes habitudes (Clean Code)
- Comprendre l'intérêt des tests (TDD)
- Installer une méthodologie

Objectifs

- Sensibiliser à quelques dispositifs qui permettent d'améliorer la qualité d'un programme ("clean code") afin qu'il réponde mieux aux attentes des utilisateurs, des clients et plus généralement des parties prenantes
- Sensibiliser à la notion de cycle de développement qui s'inscrit dans tout projet informatique de développement logiciel
- Avoir au plus vite de bonnes habitudes

Pourquoi du "Clean Code" ?

L'objectif principal du "clean code" est la construction de logiciels en minimisant le coût du changement et en maximisant la capacité à répondre au changement.



Quelques principes généraux

Objectifs :

- Suivre les conventions standards
  code, architecture, conception (les vérifier avec des outils)
- *Keep It Simple, Stupid (KISS)*
  réduire la "complexité" autant que possible
- Règle du boy scout
  laisser le code source plus propre que quand on l'a trouvé

Clean Code - Concrètement

Différents axes pour aller vers du "clean code"

- Nommer des variables en appliquant les conventions propres au langage
- Simplifier le code (ex : remplacement du while par le for)
- Ajouter des commentaires de structuration du code en vue de le découper en fonctions
- Décomposer en fonctions, fonctions auxiliaires et code principal

Clean Code - Nommage

Conventions sur le nommage

- Choisir des noms descriptifs et non ambigus
  - les noms de variables doivent refléter ce que représente la variable,
  - les noms de fonctions doivent refléter ce que fait une fonction (pas comment fait la fonction !),
  - les noms doivent être précis.
- Choisir des noms correspondant au niveau d'abstraction approprié
  - choisir des noms qui reflètent le niveau d'abstraction sur lequel vous travaillez : un nom fonctionnel au niveau fonctionnel, un nom technique au niveau technique, etc.
- Utiliser les nomenclatures du langage quand cela est possible
- Respecter les conventions "internes" (explicites dans la plupart des entreprises)

Clean Code - Nommage

- les conventions peuvent changer d'un langage à l'autre, d'une entreprise à l'autre
- il faut les connaître (savoir les trouver) et les utiliser dans un milieu professionnel.
- en Python, les noms de variables s'écrivent en minuscule et les mots sont séparés par des _ :
  `liste_a_traiter, somme_elements_liste, ...`
- dépend aussi de : qui lira/relira/retravaillera le code. En L1S2 : on écrit pour soi ET POUR LE PROF !
  Quelques exemples :
  - `lequipes`, pourrait être `liste_equipes` ou encore `equipes` (le s du pluriel pouvant être suffisant pour indiquer la liste)
  - indiquer le "i" pour les valeurs entières successives d'un compteur, d'un indice. À ne pas le mettre si ce sont des valeurs et non des indices (différence entre `membre` et `i_membre`)

Clean Code - Simplifier

Rendre le code plus lisible et compréhensible avec des boucles plus simples.

- `for` est considéré plus simple car en une seule ligne ce que le `while` fait en 3 (initialisation, condition, itération)
- le parcours des valeurs de la séquence `for elem in sequence` et considéré plus simple que le parcours d'indice `for i in range(...)` (uniquement en python).

Clean Code - Décomposer

- On commence par "la fin" : le "code principal"

```
# code principal
entrees = lecture_des_donnees()
print(fonction_de_traitement(entrees))
```

- On "remonte" en utilisant les commentaires pour structurer la décomposition

```
def fonction_de_traitement(entrees) :
    #faire ceci
    fonction_qui_fait_ceci(...)
    #faire cela
    fonction_qui_fait_cela(...)
    return ...
```


Décomposer

Ce qui donne :

```
#Definitions de fonctions utiles
def fonction_qui_fait_ceci(...) :
    # ceci

def fonction_qui_fait_cela(...) :
    # cela

def fonction_de_traitement(entrees) :
    #faire ceci
    fonction_qui_fait_ceci(...)
    #faire cela
    fonction_qui_fait_cela(...)
    return ...

# Code principal
entrees = lecture_des_donnees()
print(fonction_de_traitement(entrees))
```


Clean Code - Exemple

```
#### Code "pas clean"
L = eval(input())
r = []
j = 0
while j < len(L) :
    e = L[j]
    n,a = e[0]
    i = 1
    while i < len(e):
        p = e[i]
        if abs(p[1]-18)<abs(a-18):
            n,a = p
        i = i + 1
    r.append(n)
    j = j + 1
print(r)
```

Code (correct) qui répond au CT 2016 "Plus proche de 18 ans" : A partir d'une liste d'équipes, une équipe étant une liste de couples de membres, un membre étant un nom et un âge, afficher le membre de chaque équipe le plus proche de 18 ans.


```
#### Code "clean"
def liste_proches_de_18(lequipes):
    lproches18 = []
    #pour chaque equipe, rechercher le membre le plus proche de 18
    for eq_cour in lequipes :
        #trouver le nom du plus proche
        nom_proche = trouver_nom_proche_18(eq_cour)
        #l'ajouter
        lproches18.append(nom_proche)
    return lproches18

def trouver_nom_proche_18(equipe):
    # initialisation du plus proche de 18 avec le premier equipier
    nom_proche,age_proche = equipe[0]
    for membre in equipe[1:]:
        #si on trouve un membre plus proche, il devient le plus proche
        if abs(membre[1]-18)<abs(age_proche-18):
            nom_proche,age_proche = membre
    return nom_proche

lequipes = eval(input())
print(liste_proches_de_18(lequipes))
```

- nommage
- utilisation des for elem (et ni while, ni for i)
- décomposition

Clean Code - Exemple

Pour approfondir :

- `http://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf`
- `https://cleancoders.com/`
- Clean Code : A Handbook of Agile Software Craftsmanship de Robert C. Martin

Test Driven Development - Introduction

Voir cours sur les tests automatisés ici ou sur le moodle de l'UE Projet.

Motivations

- Remplacer la saisie manuelle, par la fourniture du jeu de tests
- Remplacer la vérification visuelle par des assertions
- Utiliser une librairie de tests 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

Test Driven Development - Introduction

Appliqué à l'exemple à l'aide de `print`, ici on prend des valeurs particulières (liste vide, cas général, cas avec une seule équipe...) :

```
1 print(liste_des_proches_de_18([])) #vide
2 print(liste_des_proches_de_18([[("Abdel",17),
    ("Bea",16), ("Calvin",21), ("Dimitri",25)],
    [("Elfie",16), ("Faustine",19),
    ("Gollum",23)], [("Hobbes",18),
    ("Ignace",19)], [("JonSnow",35)]])) #[Abdel,
    Faustine, Hobbes, JonSnow]
3 print(liste_des_proches_de_18([[("Abdel",12),
    ("Bea",13), ("Calvin",10),
    ("Dimitri",11)]])) #[Bea]
```

La limite des tests : on ne sait pas ce que le print doit nous donner à moins de le calculer.

Definition (**Wikipedia**)

Une assertion est une expression qui doit être évaluée à vrai. Si cette évaluation échoue elle peut mettre fin à l'exécution du programme, ou bien lancer une exception.

Le mot clé `assert` termine l'exécution du programme si l'assertion est fausse et affiche le message. La programmation par contrat et les tests unitaires sont basés sur les assertions.

```
1 assert PROPRIETE , "MESSAGE"
```



```
1 assert PROPRIETE , "MESSAGE"
```

Mises en garde :

- les propriétés sont calculées, attention à l'efficacité
- le programme s'interrompt si la propriété n'est pas vérifiée, cela complique l'analyse automatique
- pour le cycle "in the small", on les mettra dans un fichier à part et on y fera appel qu'en phase de mise au point.
- les asserts seront également utilisés pour les invariants (CTD suivants) UNIQUEMENT pour la mise au point

A l'aide de `assert` :

```
1 assert (liste_des_proches_de_18([]) == []), "Le␣
    resultat␣devrait␣etre␣la␣liste␣vide"
2 assert ((liste_des_proches_de_18([[("Abdel",17),
    ("Bea",16), ("Calvin",21), ("Dimitri",25)],
    [("Elfie",16), ("Faustine",19), ("Gollum",23)],
    [("Hobbes",18), ("Ignace",19)], [("JonSnow",35)]])
    == ['Abdel', 'Faustine', 'Hobbes', 'JonSnow']),
    "Le␣resultat␣devrait␣etre␣
    [Abdel,Faustine,Hobbes,JonSnow]")
3 assert ((liste_des_proches_de_18([[("Abdel",12),
    ("Bea",13), ("Calvin",10), ("Dimitri",11)]]) ==
    ['Bea']), 'Le␣resultat␣devrait␣etre␣[Bea]')
```


La fonction de tests pour la fonction `liste_des_proches_de_18` s'écrira donc :

```
1 def test_liste_des_proches_de_18():
2     assert (liste_des_proches_de_18([]) == []), "Le␣
    resultat␣devrait␣etre␣la␣liste␣vide"
3     assert ((liste_des_proches_de_18([[("Abdel",17),
    ("Bea",16), ("Calvin",21), ("Dimitri",25)],
    [("Elfie",16), ("Faustine",19), ("Gollum",23)],
    [("Hobbes",18), ("Ignace",19)], [("JonSnow",35)]])
    == ['Abdel', 'Faustine', 'Hobbes', 'JonSnow']),
    "Le␣resultat␣devrait␣etre␣
    [Abdel,Faustine,Hobbes,JonSnow]")
4     assert ((liste_des_proches_de_18([[("Abdel",12),
    ("Bea",13), ("Calvin",10), ("Dimitri",11)]]) ==
    ['Bea']), 'Le␣resultat␣devrait␣etre␣[Bea]')
```

Test Driven Development - Petit exercice : Palindrome

Exercice

Sans définir la fonction qui vérifie si une liste est un palindrome, proposer une fonction de test à l'aide d'au moins 5 assertions.

- identifier les cas particuliers, extrêmes, généraux
- faire une assertion par cas
- construire la fonction qui fait les tests

```python
# Fonction de test unitaire de la fonction est_palindrome
def test_palindrome():
    assert est_palindrome([]), "palin liste vide vrai"
    assert est_palindrome([10]), "palin singleton vrai"
    assert not est_palindrome([1,2,3]),"palin liste impaire faux"
    assert est_palindrome([1,2,1]),"palin liste impaire vrai"
    assert not est_palindrome([1,2,3,1]),"palin liste paire faux"
    assert est_palindrome([1,2,2,1]),"palin liste paire vrai"

# Fonction est_palindrome
def est_palindrome(liste):
    i = 0
    est_palin = True
    taille = len(liste)
    while est_palin and i<taille//2:
        est_palin = (liste[i] == liste[taille-i-1])
        i = i + 1
    return est_palin

# Appel a la fonction de test
test_palindrome()
```


Exercice - Sous-suite géométrique - Enoncé

Exercice

On veut extraire d'une liste LA liste dont les éléments forment une suite géométrique de raison 2. Cette liste doit être obtenue en sélectionnant successivement les éléments qui valent le double du dernier qui a été sélectionné. Le premier élément de la liste de départ est toujours sélectionné.
Exemple : `liste = [ 1, 1, 2, 6, 4, 8, 3]` donne `sousliste = [1, 2, 4, 8]`.

Exercice - Sous-suite géométrique - Enoncé

Exemple : `liste = [ 1, 1, 2, 6, 4, 8, 3]`
donne `sousliste = [1, 2, 4, 8]`.
Explication :

- ok liste[0] parce que c'est le premier
- nok liste[1] car pas double du dernier sélectionné : 1
- ok liste[2] qui vaut 2 car double du dernier sélectionné qui est 1
- nok liste[3]
- ok liste[4] car double du dernier sélectionné : 2
- ok liste[5] car double du dernier sélectionné : 4
- nok liste[6]

Dans le cas de la liste vide, le résultat est la liste vide.

Exercice - Sous-suite géométrique - TDD

- Trouver les cas particuliers, les cas extrêmes et les cas généraux en prenant des exemples d'entrées
- Proposer la fonction de test avec des assertions
- Ecrire le code du programme demandé (veiller au clean code, nommage, décomposition...)
- Votre code passe-t'il les tests que vous avez prévus ?

Exercice - Sous-suite géométrique - Couverture de test

Pour les codes erronés suivants :

- pensez-vous que votre fonction de tests identifie les erreurs ?
- quels cas avez-vous oubliés ?
- caractérisez les entrées qui font échouer le programme
- quelle est l'erreur dans le programme ?

Le pourcentage correspond au score obtenu sur PIXAL pour le code...

Exercice - Sous-suite géométrique - Couverture de test

Premier code : score 66%

```
1  Liste=eval(input())
2  SousListe=[]
3  p=0
4  SousListe.append(Liste[0])
5  a=SousListe[0]
6  for i in range (len(Liste)-1):
7     if Liste[i]==2*a:
8        SousListe.append(Liste[i])
9        a=SousListe[-1]
10 print(SousListe)
```

Premier code : commentaires

```
4 SousListe.append(Liste[0]) :
```

présuppose l'existence de Liste[0]

```
6 for i in range (len(Liste)-1) :
```

- dernière valeur de i : len(Liste)-2,
- l'élément d'indice len(Liste)-1 (qui existe si la liste n'est pas vide) ne sera pas testé.

Noms des variables pas parlants

Deuxième code : score 75%

```
1  l = eval(input())
2  c= []
3  c.append(l[0])
4  if(l == []):
5     print(c)
6  else:
7     for i in range(len(l)):
8        if((l[i] == max(c)*2)):
9           c.append(l[i])
10    print(c)
```


Deuxième code : commentaires

```
8 if((l[i] == max(c)*2)) :
```

au lieu de

```
8 if((l[i] == dernier(c)*2))...
```

Nb : dernier(L) vaut L[len(L)-1] ou L[-1]
Mais le max c'est le dernier, donc ça va ?

Troisième code : score 92%

```
1  L = eval(input())
2  sL = []
3  c1 = 0
4  if (len(L) == 0):
5    print(sL)
6  else:
7    sL.append(L[c1])
8    for i in range(len(L)):
9      if (2 * sL[c1] == L[i]):
10       c1 = c1 + 1
11       sL.append(L[i])
12   print(sL)
```

Troisième code : commentaires

- rôle de `c1` ?
- premier élément de la liste testé ?
- quel est le problème apparent ?
- pourquoi fonctionne-t-il néanmoins (apparemment) ?

Troisième code : commentaires

- Rôle de `c1` ? mémorise l'indice du dernier de `sousliste`
- Le premier élément de `L` testé est `L[0]` :

```
9            if (2 * sL[c1] == L[i]):
```

mais `sL[c1]` initialisé à `L[0]` on teste `L[0]` contre lui-même ! ! ouf, ça marche car un nombre n'est jamais le double de lui-même... sauf... si c'est 0 Cet algorithme fonctionne sur les listes qui ne commencent PAS par 0 !

- Données en entrée : une liste L d'entiers
- Sortie : `sL` est LA sous-liste géométrique extraite de L suivant l'énoncé. On abrègera en disant que `sL` est l'*extrait géométrique* de L.
- Cette étape *nomme* ce que l'on cherche et *explicite* les bornes afin de pouvoir décrire l'algorithme *en cours* d'exécution.

- Le modèle de solution est plus ou moins imposé par l'énoncé : parcourir `L` de 0 à len(L)-1 (boucle nécessaire)
  - au début de l'itération `i`, la liste `sL` doit être l'extrait géométrique du début de `L` et le traitement de l'élément suivant doit permettre d'agrandir cette zone
  - et ainsi de suite jusqu'à ce que ce début couvre toute la liste `L`.

Pour vous convaincre qu'un unique parcours suffit, faites-le *à la main* sur un exemple significatif.

Exercice - Sous-suite géométrique - Méthodologie

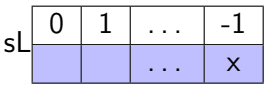
- Nécessité de préciser "début de `L`" : on utilise une *tranche* : `L[0:"où on en est"]`
- À l'itération `i` on veut que `sL` soit l'extrait géométrique de `L[0:i]`
- À la fin, on veut que `sL` soit l'extrait géométrique de `L[0:len(L)]`
- donc à la fin, on veut que `i==len(L)`
  ⇒ Condition de boucle : `while i!= len(L):`

Exercice - Sous-suite géométrique - Méthodologie

- Traiter l'élément d'indice `i` :
  - si `L[i]` vaut 2 fois le dernier[1] de `sL` on l'ajoute à la fin de `sL`
  - après ce traitement `sL` est l'extrait géométrique de `L[0:i+1]`
  - on incrémente `i`, à l'issue de quoi on peut affirmer que la liste `sL` est à nouveau l'extrait géométrique de `L[0:i]`
  - et on peut repartir pour un tour...

1. En Python, le dernier élément de `sL` est `sL[-1]`, ou `sL[len(sL)-1]`

Exercice - Sous-suite géométrique - Méthodologie

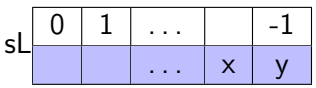


`sL` est l'extrait géométrique de `L[0:i]` (zone en vert), la case rouge l'élément d'indice `i`. Le dernier élément de `sL` est `x`.

Exercice - Sous-suite géométrique - Méthodologie

Cas où `y` vaut deux fois `x` :

- on ajoute `y` à la fin de `sL`
- on incrémente `i` (qui désigne maintenant la case suivante)



`sL` est à nouveau l'extrait géométrique de `L[0:i]` (zone en vert). Le dernier élément de `sL` est `y`.

Cas où y ne vaut pas deux fois x :

- sL est inchangée
- on incrémente i (qui désigne maintenant la case suivante)

L

0	1	...	i-2	i-1	i	...	
		...		y	z		

sL

0	1	...	-1
		...	x

sL est à nouveau l'extrait géométrique de L[0:i] (zone en vert).
Le dernier élément de sL est x.

- Quelles initialisations de i et de sL pour que sL soit dès le départ l'EG de L[0:i] ?
- i démarre à 0 ? Mais quel est l'extrait géométrique de L[0:0] ? (Non défini, il faut au moins un élément dans la liste de départ sinon le résultat est [] par convention).
- i démarre à 1 ? Parfait, car l'extrait géométrique de L[0:1] est justement la liste L[0:1]. On initialisera donc sL à L[0:1] (ou à [L[0]] c'est pareil).
- Traitement à part de la liste vide.

On écrit EG au lieu de *extrait géométrique*.

```
1 L = eval(input())
2 if (len(L) == 0):
3     # ici, [] est l'EG de L
4     sL = []
5     # ici, sL est l'EG de L
6 else:
7 # suite p. suivante
```



```
8     sL = L[0:1]
9     i = 1
10    # ici, sL est l'EG de L[0:1]
11    # et donc de L[0:i]
12    while i != len(L):
13    # ici et toujours sL est l'EG de L[0:i]
14      if L[i] == 2 * sL[-1]:
15        sL.append(L[i])
16      # ici et toujours, sL est l'EG de L[0:i+1]
17      i = i+1
18      # ici et toujours, sL est l'EG de L[0:i]
19    # ici, sL est l'EG de L[0:i] et i == len(L)
20    # donc sL est l'EG de L[0:len(L)]
21 print(sL)
```

Un peu de nettoyage (optionnel)

On peut simplifier légèrement le code ci-dessus :

```
22 L = eval(input())
23 sL = []
24 if len(L)!=0:
25     sL.append(L[0])
26     for e in L[1:]:
27         if e == 2 * sL[-1]:
28             sL.append(e)
29 print(sL)
```

Attention, la construction `L[1:]` génère une nouvelle liste, qui peut être très grande (si `L` l'est).

Développement méthodologique d'une solution

- Caractériser les entrées ⇒ Préconditions
- Caractériser l'objectif en précisant les bornes ⇒ Postconditions
- Caractériser l'itération `i` en précisant les bornes ⇒ Invariant
- Caractériser la relation entre les bornes à la fin ⇒ Condition de boucle
- Déterminer les initialisations pour assurer l'invariant à l'entrée de la boucle
- Déterminer le corps de la boucle en respectant l'invariant.

Cette méthodologie sera reprise et étoffée dans les cours/TD 3 et 4.