

## Séance 10

# Qualité de code et convention de codage



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Rappels

- Développement d'un serveur web avec le module `cherrypy`
  - Premier site web et prise en main du framework
  - Définition de routes et structure d'un site
  - Templating avec le module `jinja2`
- Création d'un **formulaire** et traitement de ses données

*Routes avec le formulaire et route de traitement*
- Définition d'une **API REST** d'accès à la BDD

*Appel d'une API avec le module `urllib`*

# Objectifs

- **Qualité** de code
  - Refactoring
  - Bonne pratique
- **Convention** de codage
  - Règle de style Pythonique
  - Mise en forme du code



Qualité de code

# Qualité de code

- Différence entre code fonctionnel et **code de qualité**

*Un code peut être audité pour évaluer sa qualité*

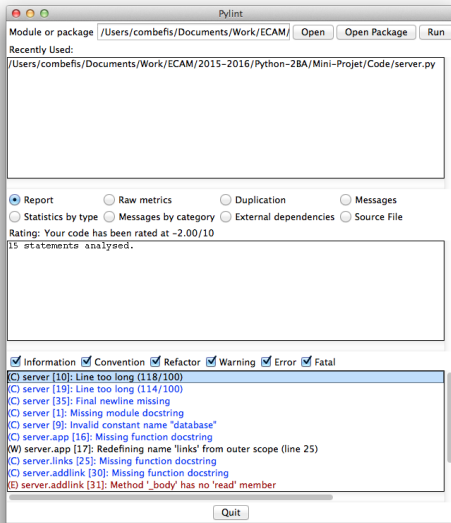
- Plusieurs **avantages**
  - Facilité de lecture du code
  - Mise à jour plus aisée
  - Moins de bug
- Ensemble de règles de bonne pratique

# Outil Pylint (1)

- **Vérification automatisée** de nombreuses règles de style
  - Convention de codage (PEP 0008)
  - Détection d'erreurs (interface, import modules...)
  - Aide au refactoring (code dupliqué...)
- Suggestion d'**amélioration** du code
- **Interface graphique** simple en Tkinter

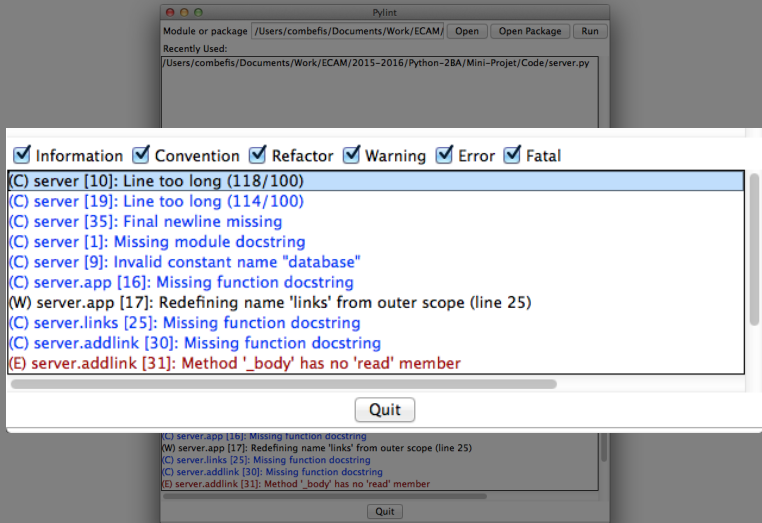


# Outil Pylint (2)





# Outil Pylint (2)





Writing programs is like writing books

# Nom de variable

- Un **nom de variable** doit expliquer sa raison d'être

*Le nom décrit la variable et la documente*

- **Longueur optimale** du nom entre 9 à 15 caractères

*Nom plus court que 9 caractères autorisé pour certains cas*

```
1 # Pas bien
2 My_AgE = 27
3 istheavatardeadornot = False
4 the_price_of_the_item = 8.99
5
6 # Bien
7 age = 25
8 dead = True
9 price = 12.99
```

# Choisir le nom

- Préférer les noms de variables en **anglais**

*Attention à l'orthographe dans les noms !*

- Attention aux noms **difficilement lisibles**

*char1 et charl, COnf et COnf, GREAT et 6GREAT...*

- Quelques situations **à éviter**

- Noms proches : `input` et `inputVal`
- Noms similaires : `clientRec` et `clientRep`
- Éviter les nombres : `total1` et `total2`
- Ne pas mélanger les langues

# Utilisation de variable

- Une variable est utilisée pour **une seule raison**

*Éviter les variables avec plusieurs signification selon la valeur*

- Éviter les nombres magiques avec des **constantes**

*Variable dont on ne change pas la valeur (et nom majuscule)*

- Utiliser une variable pour éviter de la **duplication de code**

```
1 # Pas bien
2 for beer in database['food']['beers']:
3     print(database['food']['beers'][beer]['name'])
4
5 # Bien
6 beers = database['food']['beers']
7 for beer in beers:
8     print(beers[beer]['name'])
```

# Expression booléenne (1)

- Une **condition** est une expression booléenne

*if-else pas nécessaire pour affecter une variable booléenne*

```
1 # Pas bien
2 if x > 170:
3     accepted = True
4 else:
5     accepted = False
```

- Affectation directe d'une **expression booléenne**

```
1 # Bien
2 accepted = x > 170
```

# Expression booléenne (2)

- Pas comparer une variable booléenne avec True/False

*Une variable booléenne est déjà une condition*

```
1 # Pas bien
2 if hasmoney == True:
3     print("Je peux acheter")
4
5 if forbidden == False:
6     print("Ne pas entrer")
```

- Utilisation directe de la variable booléenne

```
1 # Bien
2 if hasmoney:
3     print("Je peux acheter")
4
5 if not forbidden:
6     print("Ne pas entrer")
```

# Simplification d'expressions booléennes

## ■ Simplification avec les opérateurs logiques

*Utilisation des règles logiques sur les expressions booléennes*

## ■ Règles d'équivalence logique

$$\blacksquare \text{ not } x > 0 \equiv x \leq 0$$

$$\blacksquare \text{ not } (x \text{ and } y) \equiv \text{ not } x \text{ or } \text{ not } y \quad (\text{De Morgan})$$

```
1 # Pas bien
2 success = not grade < 10
3 result = not (temperature > 15 and temperature <= 32)
4
5 # Bien
6 success = grade >= 10
7 result = temperature <= 15 or temperature > 32
```



# Variable inutile

- Ne pas déclarer une variable qui n'est utilisée qu'une seule fois

*Sauf pour augmenter la lisibilité ou éviter un nombre magique*

```
1 # Pas bien
2 def totalprice(price, quantity):
3     taxrate = 21
4     return (price * quantity) * (1 + taxrate / 100)
5
6 sentence = "Prix total : "
7 print(sentence + str(totalprice(7.99, 5)) + " euros")
8
9 # Bien
10 TAX_RATE = 21
11
12 def totalprice(price, quantity):
13     return (price * quantity) * (1 + TAX_RATE / 100)
14
15 print("Prix total : " + str(totalprice(7.99, 5)) + " euros")
```

# Variable indexée (1)

- Utiliser à bon escient les **listes** pour avoir plusieurs variables

*Éviter les variables nommées avec des nombres*

```
1 # Pas bien
2 grade1 = 12.5
3 grade2 = 7.5
4 grade3 = 18.5
5
6 # ...
7
8 # Bien
9 grade = [12.5, 7.5, 18.5]
10
11 # ...
```

## Variable indexée (2)

- Utiliser les **boucles** pour parcourir une liste

*Permet d'effectuer le même traitement sur plusieurs données*

```
1 # Pas bien
2 # ...
3
4 print(grade1)
5 print(grade2)
6 print(grade3)
7
8 # Bien
9 # ...
10
11 for i in range(3):
12     print(grade[i])
```

# Variable indexée (3)

- Utiliser à bon escient les **listes** pour avoir plusieurs variables

*Éviter les variables nommées avec des nombres*

```
1 # Pas bien
2 grade_louis_python = 18.5
3 grade_louis_proba = 8.5
4 grade_valerian_python = 2.5
5 grade_boucquey_python = 10
6 grade_cousin_proba = 11.5
7
8 # Bien
9 grade = {
10     'theo': {'python': 18.5, 'proba': 8.5},
11     'gabby': {'python': 2.5},
12     'brieuc': {'python': 10},
13     'arnaud': {'proba': 11.5},
14 }
15
16 # ...
```

# Bloc else inutile

- Contrôler le **niveau d'indentation** en évitant des blocs else

*Affecter une valeur par défaut avant l'instruction if*

```
1  # Pas bien
2  if grade < 10:
3      verdict = "Raté"
4  else:
5      verdict = "Réussi"
6
7  # Bien
8  verdict = "Raté"
9  if grade >= 10:
10     verdict = "Réussi"
```

# Instruction return

- L'instruction return **quitte la fonction** en cours d'exécution

*Possibilité d'éviter une instruction else inutile*

```
1 # Pas bien
2 def verdict(grade):
3     if grade < 10:
4         return "Raté"
5     else:
6         return "Réussi"
7
8 # Bien
9 def verdict(grade):
10    if grade < 10:
11        return "Raté"
12    return "Réussi"
```

# Duplication de code

- Éviter la **duplication de code** (exacte ou quasi-exacte)
  - Une mise à jour du code doit se faire partout
  - Des erreurs de copier-coller sont possibles
  - Le code est lourd

```
1 # Pas bien
2 dvd = 17.50
3 coca = 0.70
4 print("Prix : ", dvd * 1.21)
5 print("Prix : ", coca * 1.21)
6
7 # Bien
8 def price(item):
9     return item * 1.21
10
11 items = {'dvd': 17.50, 'coca': 0.70}
12 for item in items:
13     print("Prix : ", price[item])
```

# Découpe en fonctions

- Éviter des **codes trop longs** en créant des fonctions

*Simplifier la lisibilité et la réutilisabilité*

```
1  # Pas bien
2  data = (12, 7)
3  if data[0] < data[1]:
4      sorted = data
5  else:
6      sorted = (data[1], data[0])
7
8  # Bien
9  def sort(data):
10     if data[0] > data[1]:
11         return (data[1], data[0])
12     return data
13
14 sorted = sort((12, 7))
```



# Layout de code

- Un **layout** correct de code fait ressortir la logique

*Utilisation de blancs, lignes vides, parenthèses pour expressions...*

- Une ligne de code ne devrait pas dépasser **80 caractères**

- Dans une classe, d'abord les constructeurs puis les méthodes

```
1  # Pas bien
2  x=12+3*y
3  if x>=0:print("Hello, it's me")
4  else:print("Heeeeey macarena!")
5
6  # Bien
7  x = 12 + 3 * y
8  if x >= 0:
9      print("Hello, it's me")
10 else:
11     print("Heeeeey macarena!")
```

# Commentaire (1)

- On retrouve **plusieurs types** de commentaires
  - Répétition du code
  - Explication du code
  - Marqueur dans le code
- Un **bon commentaire** peut servir plusieurs buts
  - Résumé du code (focus sur le pourquoi, pas le comment)
  - Expliquer l'intention du code (niveau problème, pas solution)
  - Information additionnelle non présente dans le code

“Good code is its own documentation” — Steve McConnell

# Commentaire (2)

727

```
// drunk, fix later
```

Wish I were kidding. And knowing the developer who wrote the code, I think he meant it literally.

share

edited Oct 9 '08 at 18:02

community wiki  
3 revs, 2 users 83%  
Daniel Papasian

720

```
// Magic. Do not touch.
```

share

answered Oct 8 '08 at 22:07

community wiki  
Jason Sundram

637

```
return 1; # returns 1
```

share

answered Oct 8 '08 at 23:13

community wiki  
Lateral

355

```
long long ago; /* in a galaxy far far away */
```

share

edited Mar 16 '11 at 18:14

community wiki  
2 revs, 2 users 67%  
Juliano



Convention de codage

# Convention de codage

- Ensemble de **règles de style** de codage

*Conventions établies pour une société, un langage, un projet...*

- Plusieurs **avantages**
  - Un développeur rentre plus vite dans le code d'un autre
  - Uniformité pour un projet avec plusieurs développeurs
  - Augmentation de la lisibilité et de la productivité
  - Consistance accrue au sein d'un projet

# Code pythonique

- **PEP 0008** — Style Guide for Python Code

*<https://www.python.org/dev/peps/pep-0008/>*

- Un code est **plus souvent lu** qu'écrit (Guido van Rossum)

*Ensemble de conventions pour améliorer la lisibilité*

- Il s'agit d'un **guide** et pas d'une Bible

*À ne pas suivre aveuglément, faire confiance à son jugement*

# Mise en page (1)

- Indentation du code avec **quatre espaces**
- **Alignement** des éléments imbriqués en cas de coupure de ligne

```
1 def createperson(firstname, lastname, birthdate,  
2                 sex, address, weight, height):  
3     if (sex == 'F' or  
4         sex == 'M'):  
5         # ...
```

- Déclaration de **liste**, ensemble... sur plusieurs lignes

```
1 data = [  
2     1, 2, 3,  
3     4, 5, 6  
4 ]
```

# Mise en page (2)

- Longueur maximale d'une ligne de code limitée à **79 caractères**

*Et 72 pour des longs blocs de texte (commentaire, docstring...)*

- **Continuation de ligne** avec le caractère backslash

```
1 with open('/Users/combefis/Desktop/video-ole-ole.mp4'), \  
2     open('/Users/combefis/Desktop/vchenet-brouette.jpg'):  
3     # ...
```

- **Lignes vides** avant les définitions de fonction et classe

- Deux lignes vides pour celles dans le fichier
- Une ligne vide avant les méthodes
- Et séparation de parties logiques



# Importation

- Un **import** par ligne pour les modules

*Éviter les `from ... import *`*

```
1 # Pas bien
2 import os, sys
3
4 # Bien
5 import os
6 import sys
```

- Après commentaires de module, avant constantes globales
  - 1 d'abord librairie standard
  - 2 puis librairies 3rd party
  - 3 et enfin spécifique à l'application

# Espace et blanc

- Éviter les espaces dans les situations suivantes
  - Juste dans les parenthèses, crochets, accolades
  - Avant une virgule ou un deux-points (sauf slices)
  - Avant une parenthèse d'appel de fonction, crochet d'accès
  - Pas plus d'un espace autour opérateur (sauf param. nommé)

```
1 # Pas bien
2 def func(name = 'VCH'):
3     preferences = { name      : ['CBF'      , 'FLE'    ]}
4     values      = data      [2 :4]
5     verdict     = getresult ('LUR')
6
7 # Bien
8 def func(name='VCH'):
9     preferences = {name: ['CBF', 'FLE']}
10    values = data[2:4]
11    verdict = getresult('LUR')
```

# Commentaire

- Mettre à jour les commentaires en même temps que le code
- Écrire les commentaires en anglais

*Faire une phrase complète qui finit par un point*

- Éviter les commentaires sur la même ligne

*Distrait le lecteur du code*

```
1  # Pas bien
2  x += 1          # Incrémente la variable x de une unité
3
4  # Bien
5  # Compensation pour la largeur de la bordure
6  x += 1
```

# Choisir ses noms

- Nom de variables et de fonctions en **minuscule**
- Nom de constante en **majuscule**
- Nom de classe en **majuscule** en casse chameau

```
1 class ComplexNumber:
2     def __init__(self, real, imag):
3         self.__real = real
4         self.__imag = imag
5
6     def norm(self):
7         return math.sqrt(self.__real ** 2 + self.__imag ** 2)
8
9 ZERO = ComplexNumber(0, 0)
```

# Crédits

- <https://www.flickr.com/photos/dreamsjung/12613244714>
- <https://www.flickr.com/photos/alixia88/6777232356>
- <https://www.flickr.com/photos/roadsidepictures/2222477980>