Les bonnes pratiques de codage en Python

Python Enhancement Proposal

Denis Arrivault[1] François-Xavier Dupé[2] Dominique Benielli[1]

[1]Laboratoire d'Excellence Archimède
Aix Marseille Université

[2]Laboratoire d'Informatique et Systèmes
Aix Marseille Université

Formation Python Scientifique

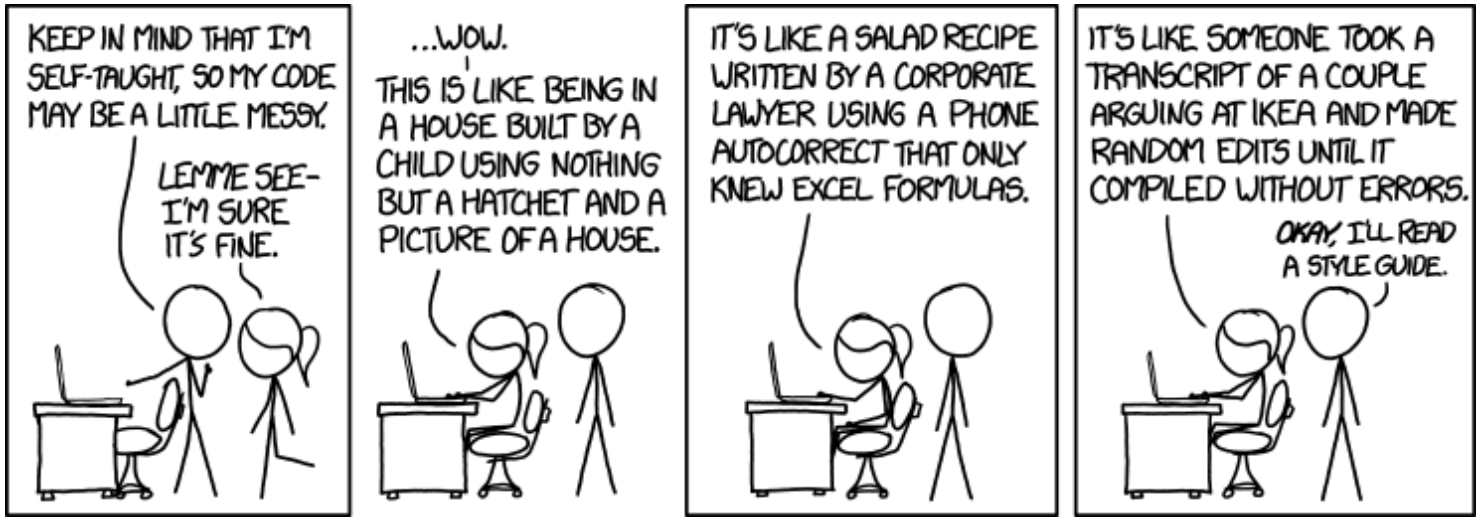
Outline

Outline

Pourquoi des « bonnes pratiques » ?

Pourquoi des « bonnes pratiques » ?

FIGURE – xkcd (Creative Commons)



Outline

C’est quoi un PEP ?

C'est quoi un PEP ?

un document

- pour fournir des infos à la communauté Python
- pour décrire des nouvelles fonctionnalités Python
- ...

officiel

- Revue, discutée, approuvée, intégrée
- ... ou rejetée par la communauté.

C'est quoi un PEP ?

Trois types de PEP

- Standards Track PEP : nouvelles fonctionnalités.
- Informational PEP : problèmes de design, guides, infos.
- Process PEP : comme les Standards Tracks mais pour tout ce qui n'est pas du code.

Au programme

- PEP 20 (Informational) : The Zen of Python
- PEP 8 (Process) : Style Guide for Python Code
- PEP 257 (Informational) : Docstring Conventions

Outline

La philosophie du Zen : PEP 20

La philosophie du Zen : PEP 20

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity .
- Errors should never pass silently.
- Unless explicitly silenced.

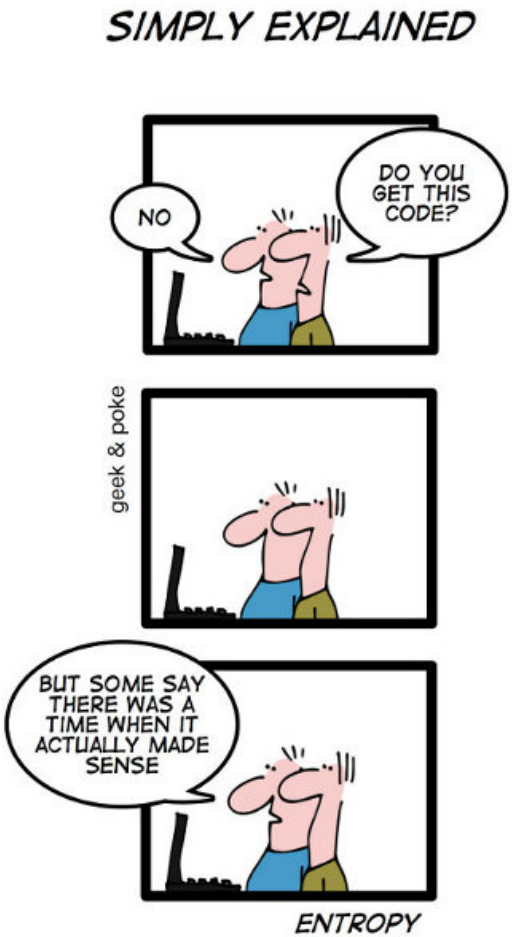
La philosophie du Zen : PEP 20

- In the face of ambiguity, refuse the temptation to guess.
- There should be one – and preferably only one – obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those.

Outline

Les règles : PEP 8

FIGURE – Geek & Poke (Creative Commons)



Contenu

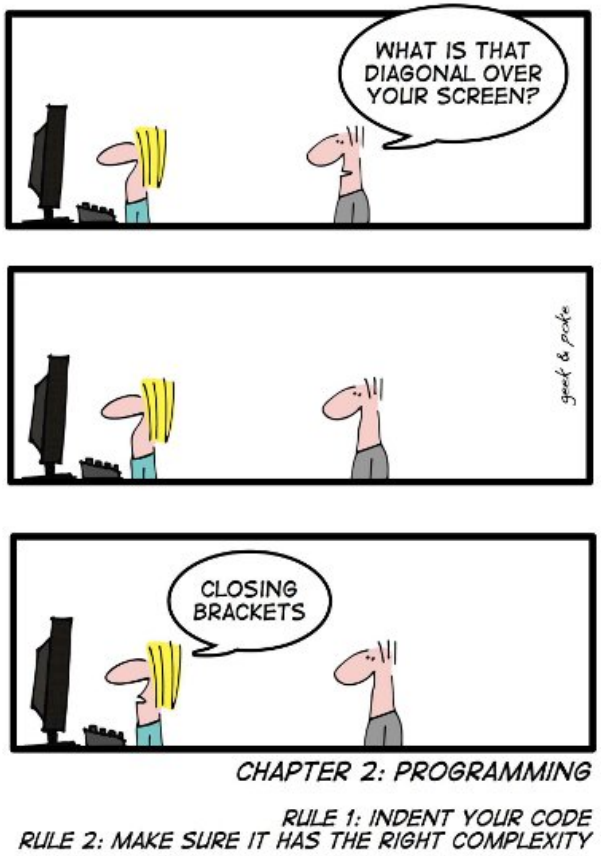
- Disposition du code
- Espaces dans les expressions
- Commentaires
- Compatibilité des versions
- Conventions de nommage
- Recommandations de programmation

Avec une mise en garde :

> *A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.*
>
> *But most importantly : know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask !*

Disposition du code

FIGURE – Geek & Poke (Creative Commons)



Disposition du code

- Indentation : 4 espaces, éviter les tabulations.
- 80 (79 sans le retour chariot) caractères max par ligne (72 pour les textes de commentaires) : correspond à la ligne verticale de spyder.
- On ne coupe pas une ligne n'importe où (après un opérateur pas avant).
- 80 caractères $\Rightarrow$ limiter la complexité.

Disposition du code

```
# Oui :
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Oui : sans argument sur le première ligne
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)

# Non : arguments sur la première ligne sans alignement
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

Disposition du code

```python
# Oui :
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Non : confusion des arguments avec les instructions
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Disposition du code

```
# Non : même si l'indentation est respectée, il y a confusion
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# Oui :
if (this_is_one_thing and
    that_is_another_thing):
    # Un commentaire pour séparer.
    do_something()

# Oui :
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

Disposition du code

```
# Les fermetures de () ou [] multi-lignes s'alignent
# avec le premier caractère non blanc de la ligne du dessus :
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
# ou de la première ligne de déclaration :
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Disposition du code

```
#On coupe les lignes après les opérateurs :
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Disposition du code

Espacements

- Deux lignes vides entre éléments 'top-niveau' (classes)
- Une ligne entre les éléments 'internes' (méthodes)

Encodage

- Python 2 utilisent l'encodage ASCII pour la lecture des sources.
- Python 3 utilisent l'encodage utf-8.
- Pour utiliser du utf-8 en python 2, il faut ajouter en début de fichier (spyder l'ajoute par défaut même en python 3) :

```
#-*- coding: utf-8 -*-
```

- L'utilisation de l'anglais est recommandé.

Disposition du code

l'importation des modules

- Au début du fichier ;
- Après les commentaires / docstrings
- Chaque module doit être importé sur une ligne différente sauf en cas de hiérarchie.

NON :

```
import os, sys
```

OUI :

```
import os
import sys
from subprocess import Popen, PIPE
```

Disposition du code

l'importation des modules

- Eviter les importations génériques.
- On peut spécifier le ou les objets à importer.
- Mais en cas de conflits il faut utiliser les noms entiers.

NON :

```
from mypkg import *
```

OUI :

```
from myclass import TheClass
```

```
import myclass
import foo.bar.yourclass

my_object = myclass.TheClass()
your_object = foo.bar.yourclass.TheClass()
```

Disposition du code

Ordre des importations

1. Bibliothèque standard
2. Bibliothèque(s) tierce(s)
3. Mes bibliothèques locales

- Une ligne blanche entre chaque groupe d'importation.

Espaces dans les expressions

Éviter les espaces superflus.

Pas d'espace

- dans les parenthèses, crochets ou accolades,
- avant une virgule, un point-virgule, deux points,
- avant une parenthèse ouvrante commençant une liste d'arguments de fonctions,
- avant un crochet ouvrant sur une indexation.

Des espaces

- autour des opérateurs.

Espaces dans les expressions

NON :

```
spam( ham[ 1 ], { eggs: 2 } )
if x == 4 : print( x , y ) ; x , y = y , x


spam (1)
dict ['key'] = list [index]
x              = 1
y              = 2
long_variable = 3
x = x * 2 - 1
```

Espaces dans les expressions

Oui :

```
spam(ham[1], {eggs: 2})
if x == 4:
    print(x, y)
    x, y = y, x
spam(1)
dict['key'] = list[index]
x = 1
y = 2
long_variable = 3
x = x*2 - 1
```

Commentaires

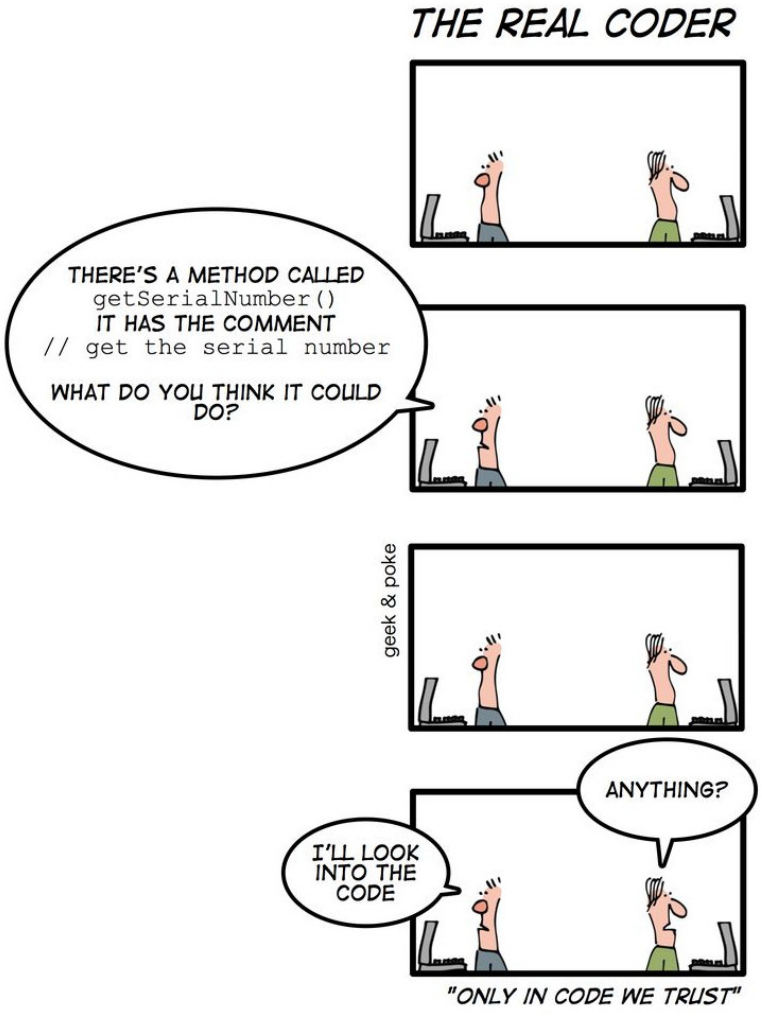


FIGURE – Geek & Poke (Creative Commons)

Commentaires

De manière générale :

- Des commentaires qui contredisent le code sont pires que pas de commentaire ;
- Garder les commentaires à jour quand le code change ;
- Faire des phrases complètes commençant par des majuscules à moins que qu'elles ne commencent par un identifieur en minuscules ;
- Ne jamais changer la casse d'un identifieur ;
- Ecrire les commentaires en anglais à moins d'être sur à 120% que le code ne sera lu que par des francophones.

Commentaires

Pour les blocs

- S'appliquent sur le code qui suit ;
- Chaque ligne commence par un "#" suivi d'un espace ;
- Les paragraphes sont séparés d'une ligne vide commençant par un "#".

Pour les commentaires de ligne

- Séparés par deux espaces après le code ;
- Commencent par un "#" suivi d'un espace ;
- Doivent se justifier, éviter d'enfoncer des portes ouvertes :

```
x = x + 1   # x incrementing
```

Pour les Documentation Strings

PEP 257

Conventions de nommage

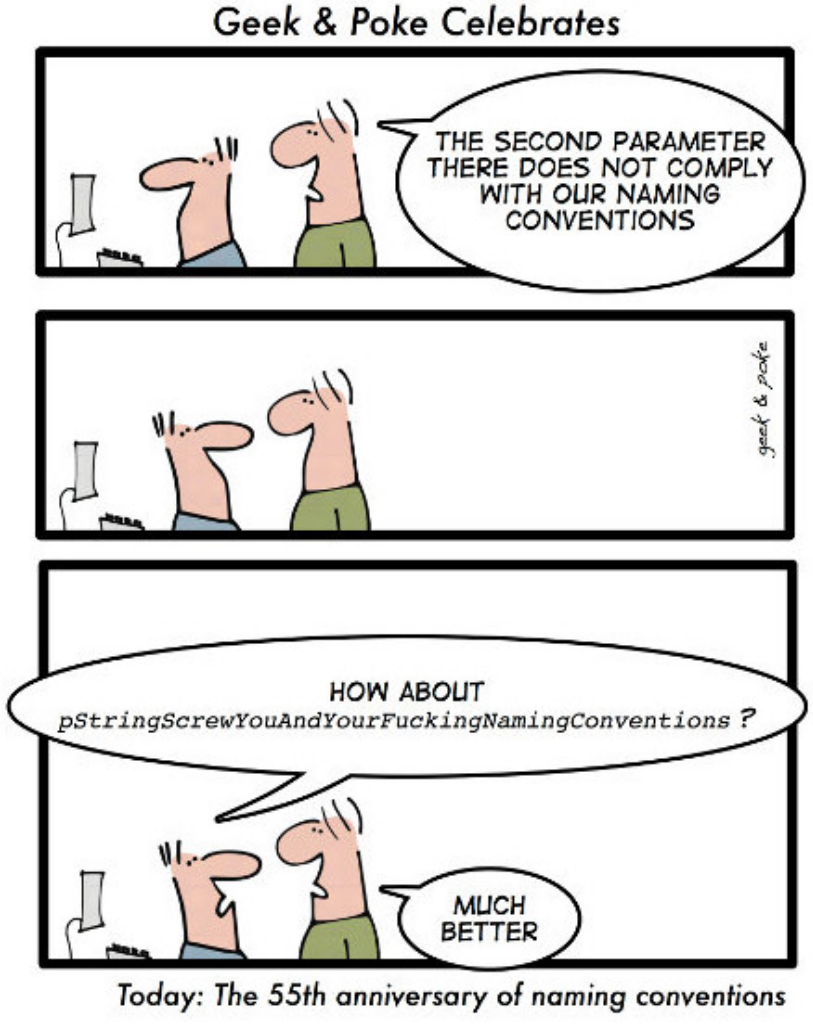
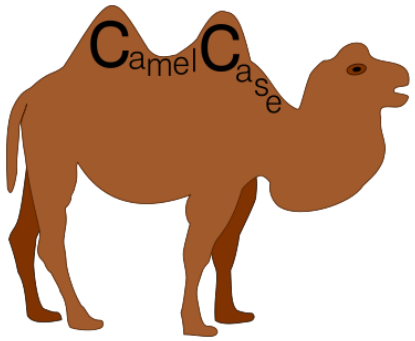


FIGURE – Geek & Poke (Creative Commons)

Conventions de nommage

Principe général : les noms reflètent l'usage plutôt que l'implémentation.

Styles existants distinctifs



- b (une seule minuscule)
- B (une seule majuscule)
- minuscules
- MAJUSCULES
- minuscules_avec_des_underscores
- MAJUSCULES_AVEC_DES_UNDERSCORES
- MotsEnMajuscule (appelé souvent CamelCase)
- motAvecCapitalisationMelangee (minuscule au premier mot)
- Mots_En_Majuscule_Avec_Des_Underscores (Moche !)

Conventions de nommage

Conventions prescrites

- **modules** : `module` (court, minuscule)
- **classes** : `CorrectClassName`
- **exceptions** : `IncorrectClassNameError`
- **fonctions** : `get_correct_number(random = False)`
- **méthodes** : `get_correct_number(self)`
- **variables** : `number = my_object.get_correct_number()`
- **constantes** : `ANSWER_TO_LIFE_UNIVERSE = 42`

Conventions proscrites

Ne jamais utiliser les caractères 'l'(l minuscule), 'O'(o majuscule) ou 'I'(i majuscule) comme nom de variable. Avec certaines fontes elles se confondent avec 0 ou 1.

Conventions de nommage

Underscore de début et de fin

- **1 au début** : `_ma_variable_interne`
- **1 à la fin** : `ma_variable_en_conflit_avec_une_variable_python_`
- **2 au début** : `__ma_méthode_privée`
- **2 au début et à la fin** = **Interdit !** Objets magiques de Python : `__init__`, `__import__`, `__file__`...

Fortement conseillé

Déclarer les API publiques d'un module à l'aide de la variable `__all__` (usuellement dans le fichier `__init__.py`).

Recommandations de programmation

Pour les booléens :

- Ne pas utiliser :

```
if my_bool == True:
```

- Ni :

```
if my_bool is True:
```

- Utiliser :

```
if my_bool:
```

Pour les autres types :

- Ne pas utiliser :

```
if x:
```

- Utiliser :

```
if x is not None:
```

- Ne pas utiliser :

```
if not len(seq):
```

- Utiliser :

```
if not seq:
```

Recommandations de programmation

Les exceptions

- Préciser ce qui a causé l'exception (“Erreur” n'est pas un message suffisant. . .).
- En python 3, utiliser les exceptions chainées.

```
class NetworkError(Exception):
    """Network error base class."""
    pass
...
raise NetworkError("Cannot fin host.")
```

Recommandations de programmation

Les exceptions

- Les `except` doivent être spécifiques autant que faire se peut.
- Le bloc `except` ne doit comporter que le code strictement nécessaire (pas de commentaire).
- Ne pas utiliser `except:` seul pour l'interception générique mais `except Exception:`

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
except Exception:
    raise
else:
    return value
```

Recommandations de programmation

Les chaînes de caractères

- Ne pas utiliser les méthodes du module `string`. Elles sont maintenant toutes disponibles directement comme méthodes des instances de `str`.
- Utiliser `endswith` ou `startswith` plutôt que les slices

```
if foo.startswith('bar'):
    print "good"

if foo[:3] == 'bar':
    print "not good"
```

Vérifier la conformité pep8

En ligne de commande

- Installer `pep8` avec `pip install pep8`.
- Lancer l'analyseur sur un fichier avec `pep8 foo.py`

Avec Spyder et Anaconda

- Installer `pep8` avec `Anaconda\Scripts\pip.exe install pep8`.
- Dans textitOutils/Préférences/Editeur/Introspection et analyse de code, cocher la case *Analyse de style pep8*.

Outline

Les Docstring : PEP 257

FIGURE – David Holt (Creative Commons)

Définition

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` *special attribute of that object.*

- Une chaîne de caractère que l'on n'assigne pas.
- Elle est placée à des endroits spécifiques du code : juste après la définition d'une fonction, d'une classe ou en début de module.
- Elle documente l'objet et correspond à ce que renvoie sa commande `help()`.

Où ?

Obligatoire dans :

- tous les modules publics
- toutes les fonctions
- toutes les classes
- toutes les méthodes de ces classes

Non nécessaire dans :

les méthodes non publiques où il peut être remplacé par un simple commentaire après la ligne `def`.

Comment ?

Propriétés

- Utiliser les *"""trois guillemets anglais"""*.
- Si le docstring contient des backslashes, utiliser la syntaxe *r"""mon docstring avec \"""*.

Docstring sur une ligne

- Pour préciser l'effet d'une fonction ou méthode.
- Elle doit commencer par "Do this" ou "Return that". Pas de description.

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Comment ?

Docstring sur plusieurs lignes

- Elle commence par une ligne simple contenant une précision (comme pour le Docstring sur une ligne).
- Puis il faut une ligne blanche.
- La suite résume précisément les différentes i/o et/ou l'usage.

Exemple

```python
# -*- coding: utf-8 -*-
"""
This is a super module called mymodule.py
@author: arrivault
"""


def create_complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)

    Returns:
        complex - real + imag*j
    """
    if imag == 0.0 and real == 0.0:
        return complex(0, 0)
    return complex(real, imag)
```

Exemple

```
>>> import mymodule
>>> help(mymodule)
```

```
Help on module mymodule:

NAME
    mymodule

DESCRIPTION
    This is a super module called mymodule.py
    @author: arrivault

FUNCTIONS
    create_complex(real=0.0, imag=0.0)
        ...
FILE
    /path_to/mymodule.py
```

Exemple

```
>>> import mymodule
>>> help(mymodule.create_complex)
```

```
Help on function create_complex in module mymodule:

create_complex(real=0.0, imag=0.0)
    Form a complex number.

    Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)

    Returns:
        complex - real + imag*j
```

Outline

doctest

Principe

- Rendre executables les exemples de code fournis dans les Docstring.
- On écrit les exemples sous forme de commandes Python (avec les chevrons).
- On précise à l'execution du module que les exemples sont executables avec les instructions :

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Remarques

- Les doctest ne remplacent pas les tests unitaires !
- Un doctest sert principalement à valider l'exemple donné (il n'est pas exhaustif).

doctest

Exemple

```
def create_complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)

    Returns:
        complex - real + imag*j

    Example:

    >>> a = create_complex(1,2)
    >>> print(a)
    (1+3j)
    """
    if imag == 0.0 and real == 0.0:
```

doctest

Exemple

Pour lancer le test il suffit d'exécuter le module :

```
& python monmodule2.py
**********************************************************************

File "mymodule2.py", line 22, in __main__.create_complex
Failed example:
    print(a)
Expected:
    (1+3j)
Got:
    (1+2j)
**********************************************************************

1 items had failures:
   1 of   2 in __main__.create_complex
***Test Failed*** 1 failures.
```

unittest

Principe

- Le module *unitest* est un module qui permet d'effectuer des tests unitaires systématiques.
- Pour chaque module de code on écrit un module de tests associés.
- Dans un module de tests, les tests sont les méthodes d'une classe héritant de *unittest.TestCase*
- L'instruction d'exécution d'un module de tests s'écrit :

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

unittest

Exemple

```
# -*- coding: utf-8 -*-
"""
Module de test de mymodule
@author: arrivault
"""
import unittest
import random
from mymodule import create_complex


class TestMymodule(unittest.TestCase):
    def test_zero(self):
        self.assertEqual(create_complex(), complex(0, 0),
                         "Test d'initialisation")
```

unittest

Exemple

```
    def test_random(self):
        r = 100 * (random.random() - 0.5)
        i = 100 * (random.random() - 0.5)
        self.assertEqual(create_complex(r, i), complex(r, i), "
            Test random")

    def test_fail(self):
        self.assertEqual(create_complex(1, 2), complex(1, 3),
                         "Test faux pour l'exemple")

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

unittest

Exemple

Pour lancer le test il suffit d'exécuter le module :

```
& python test_mymodule.py
F..
======================================================================

FAIL: test_fail (__main__.TestMymodule)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "test_mymodule.py", line 21, in test_fail
    self.assertEqual(create_complex(1,2),complex(1,3),"Test faux
        pour l'exemple")
AssertionError: (1+2j) != (1+3j) : Test faux pour l'exemple

----------------------------------------------------------------------

Ran 3 tests in 0.000s

FAILED (failures=1)
```

Outline

Mesurer la qualité

Mesurer la qualité

Il existe de nombreuses métriques permettant de mesurer la qualité d'un code :

- Le nombre de lignes de code.
- Le ratio de lignes de commentaires, lignes blanches.
- La complexité de McCabe (ou complexité cyclomatique).
- Les métriques de Halstead.
- L'index de maintenabilité.
- ...

La couverture

- Pourcentage du code couvert par les tests.
- Il faut se fixer un taux le plus proche possible de 100%.
- On pourra utiliser *coverage* avec *nosetests* ou *pytest-cov* avec *pytest*

Outline

Les Fonctions

Portées des variables

Passage des paramètres

- Si les paramètres passés à une fonction sont de type mutable alors ils peuvent être modifiés par la fonction (= passage par référence). C'est le cas pour les listes et les dictionnaires.
- Si les paramètres passés à une fonction sont de type non mutable alors ils ne peuvent pas être modifiés par la fonction (= passage par valeur). C'est le cas pour les booléens, les entiers, les flottants, les complexes, les strings, les tuples.

Portée des variables

- Les variables définies dans une fonction ont une portée locale et sont détruites à la fin de l'appel.
- Pour modifier dans une fonction, une variable définie hors de la fonction, il convient d'utiliser le mot clef `global`.

Portées des variables

Exemple

```
# -*- coding: utf-8 -*-
# file portees_var.py


a = 1
b = 2
c = 3


def incr_affiche(b):
    global c
    # a += 1
    # => UnboundLocalError
    b += 1
    c += 1
    print("Dans la fonction")
    print(a, b, c)

print("Avant l'appel")
print(a, b, c)
incr_affiche(b)
print("Après l'appel")
print(a, b, c)
```

Portées des variables

Exemple

```
$ python portees_var.py
Avant l'appel
1 2 3
Dans la fonction
1 3 4
Après l'appel
1 2 4
```

Les fonctions anonymes

L'opérateur lambda

Des fonctions anonymes de la forme d'une expression peuvent être crées en utilisant l'opérateur *lambda* sous la forme :

lambda args : expression

Exemple :

```
>>> sum = lambda x, y : x + y
>>> sum(5,6)
11
```

Les comprehensions de liste

Déclaration

Sans condition :

```
[function(items) for items in s]
```

Avec conditions :

```
[function(items) for items in s if <conditions>]
```

- Une compréhension de liste consiste à appliquer `function` à tous les éléments de la liste `s`. Une nouvelle liste est créée en retour.
- On peut spécifier des conditions sur les éléments de `s` avec une instruction conditionnelle `if`.
- Elle remplace progressivement les fonctions `map` et `filter` qu'il est conseillé de ne plus utiliser.

Les comprehensions de liste

Sans condition

```
>>> [x * x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Les comprehensions de liste

Sans condition

```
>>> [x * x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Avec conditions

```
>>> f = [(1,2), (3,4), (4,5)]
>>> g = [x + y for x,y in f if x % 2]
>>> g
[3, 7]
```

Les comprehensions de liste

Sans condition

```
>>> [x * x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Avec conditions

```
>>> f = [(1,2), (3,4), (4,5)]
>>> g = [x + y for x,y in f if x % 2]
>>> g
[3, 7]
```

Avec compositions

```
>>> [x+y for x,y in [(x,10+x) for x in range(10)] if x % 2]
```

A votre avis ?

Les comprehensions de liste

Sans condition

```
>>> [x * x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Avec conditions

```
>>> f = [(1,2), (3,4), (4,5)]
>>> g = [x + y for x,y in f if x % 2]
>>> g
[3, 7]
```

Avec compositions

```
>>> [x+y for x,y in [(x,10+x) for x in range(10)] if x % 2]
[12, 16, 20, 24, 28]
```