



**HAL**  
open science

# Tests & Documentation - Les bonnes pratiques de la programmation avec Python

Stéphane Guinard

► **To cite this version:**

Stéphane Guinard. Tests & Documentation - Les bonnes pratiques de la programmation avec Python. École d'ingénieur. France. 2018. cel-02182414

**HAL Id: cel-02182414**

**<https://hal.science/cel-02182414>**

Submitted on 12 Jul 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

**Tests**  
**Documentation - Les bonnes pratiques de la**  
**programmation avec Python**

Stéphane Guinard

► **To cite this version:**

Stéphane Guinard. Tests

Documentation - Les bonnes pratiques de la programmation avec Python. École d'ingénieur. France. 2018. cel-02182414

**HAL Id: cel-02182414**

**<https://hal.archives-ouvertes.fr/cel-02182414>**

Submitted on 12 Jul 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Tests & Documentation

Les bonnes pratiques de la programmation avec Python

---

Division des enseignements en informatique

Stéphane Guinard

# Documentation

---

# Pourquoi documenter son code

- Expliquer ce que font les fonctions, classes, modules...
- **Indispensable** pour rendre le code exploitable par d'autres

```
def ma_fonction(a, b):  
    """  
    Ligne générale de description  
  
    Description plus détaillée, si besoin, pour  
        expliquer comment la fonction  
    fait ce qu'elle fait, de ce qu'elle utilise...  
  
    :param a: description de ce que contient a  
    :type a: type de la valeur attendue dans a  
    :param b: description de ce que contient b  
    :type b: type de la valeur attendue dans b  
    :return: description de ce que retourne la  
        fonction  
    :returntype: type de la valeur retournée par la  
        fonction  
    """
```

# Doctrings vs Commentaires

Les commentaires sont complètement ignorés par l'interpréteur Python tandis que les docstrings sont chargées et gardées en mémoire.

Docstrings :

```
"""
    Docstrings
"""
'''
    Docstrings
'''
```

Commentaires :

```
# Commentaire
```

Après le prototype de la fonction :

```
"""  
    Résumé sur un seule ligne.  
  
    Contenu détaillé  
    sur plusieurs lignes.  
"""
```

Les balises :

```
"""  
    :param arg1: description de l'argument 1  
    :type arg1 : type de l'argument 1  
    :param arg2: description de l'argument 2  
    :type arg1 : type de l'argument 2  
    :return: description de la valeur de retour  
    :rtype: type de la valeur de retour  
"""
```

# Exemple

```
def addition (a, b):  
    """  
    Addition de deux nombres.  
  
    Cette fonction permet d'additionner deux  
        nombres réels et retourne également un  
        nombre réel. Elle est équivalente à l'opé  
        rateur '+'.  
  
    :param a: premier nombre à additionner  
    :type a: float  
    :param b: deuxième nombre à additionner  
    :type b: float  
    :return: résultat de l'addition  
    :rtype: float  
    """  
    return a + b
```



- La fonction `help` :

```
>>> help(math.cos)
Help on built-in function cos in module math:

cos(...)
    cos (x)

    Return the cosine of x (measured in radians).
```

- L'attribut "`__doc__`" :

```
>>> math.cos.__doc__
'cos(x)\n\nReturn the cosine of x (measured in
radians).'
```

# Tests

---

C'est une pratique courante en programmation :

- Les **tests unitaires** permettent de tester de petites portions de code (une fonction, une classe, un module) indépendamment du reste du programme.
- Les **tests fonctionnels** permettent de valider les fonctionnalités d'une application d'un point de vue utilisateur. Ils permettent de valider l'enchaînement des briques élémentaires.
- Les **tests de performances** sont constitués de tout un ensemble de tests liés à la performance de l'application : test de charge, tests de performance en temps de réponse, tests aux limites...

Nous nous intéresserons dans ce cours uniquement à la mise en œuvre de **tests unitaires**.

## Concrètement

Un test unitaire est un morceau de code permettant d'en soumettre un autre et d'analyser le résultat obtenu.

Pourquoi faire des tests unitaires :

- s'assurer que l'application fonctionne
- tester les cas limites
- après modification/ajout de code, s'assurer que l'application fonctionne *toujours*

Les bonnes pratiques :

- tester la réussite, l'échec et la cohérence
- rechercher la difficulté, proscrire les tests triviaux
- écrire des tests déterministes
- séparer le code des tests du code de l'application

**N'hésitez pas à écrire plusieurs tests par fonction !!**

# Exemple 1

```
def factorielle(n):
    if type(n) != type(int):
        print("Error: n doit etre
              un entier")
        return False
    if n < 0:
        print("Error: n doit etre
              >= 0")
        return False
    if n > 10000:
        print("Error: n est trop
              grand")
        return False

    if n == 1 or n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

```
# Tests
>>> [factorielle(n) for n
      in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorielle(-1)
Error: n doit etre >= 0
>>> factorielle(1.5)
Error: n doit etre un
      entier
>>> factorielle(1e100)
Error: n est trop grand
```

## Exemple 2

Si l'on est pas sur de la présentation du résultat.

```
def divide_par_un(n):
    """
    Retourne la division d'un nombre entier par 1.
    """
    if type(n) != type(int):
        print("Erreur : n doit etre un entier")
        return False
    return n / 1

# Tests
>>> divide_par_un(4)
4.0
# On s'attendait a "4" :-(
>>> divide_par_un(4) == 4
True
# On s'attendait a True :-)
>>> divide_par_un(1.5)
Erreur: n doit etre un entier
```

Il existe des modules permettant de faciliter la mise en œuvre de tests unitaires : doctest. Ça nous permet notamment d'écrire les tests directement dans la documentation des fonctions.



A l'exécution, ça ressemble à ça :

```
*****  
  
File "__main__" , line 4 , in __main__ . addition  
Failed example:  
addition (0 , 1)  
Expected:  
0  
Got:  
1  
*****  
  
1 items had failures:  
1 of  
2 in __main__.addition  
*** Test Failed *** 1 failures.  
TestResults (failed =1, attempted=2)
```