PEP8 – PEP257 en français (extrait), documentation riche

18 mars 2014

Ceci est un extrait des PEP8 PEP257 originaux qui reprend les points importants pour l'apprentissage des bonnes méthodes de programmation en Python pour cette première année. Les auteurs originaux sont Guido van Rossum et Barry Warsaw

1 Introduction

L'importance de la cohérence

L'une des clefs du raisonnement de Guido est que le code est plus souvent lu qu'écrit. Les indications données ici ont pour finalité d'améliorer la lisibilité du code et de rendre plus cohérent tout le code Python. Comme le dit la PEP 20 (accessible également par la commande `import this`), la "lisibilité compte" :

> The Zen of Python, by Tim Peters
> 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.
> 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 !

Un guide de style est proche de la cohérence. La cohérence avec ce style est important. La cohérence au sein d'un projet l'est encore plus. La cohérence au sein d'un module ou d'une fonction l'est encore plus.

Mais le plus important : savoir quand le code n'est pas cohérent ; parfois le guide de style n'est tout simplement pas appliqué. Lorsque vous avez un doute, remettez-vous-en à votre meilleur jugement. Regardez d'autres exemples et décidez ce qui semble le mieux. N'hésitez pas à demander !

Il y a deux bonnes raisons pouvant mener au non-respect d'une règle :

— Lorsque appliquer la règle revient à rendre le code moins facile à lire, même pour quelqu'un habitué à lire du code respectant la règle.
— Être incohérent avec du code environnant, ce qui du coup le casse.

2 Présentation du code

2.1 Indentation

Utiliser 4 espaces par niveau d'indentation.

Ne jamais mélanger tabulations et espaces.

La méthode la plus populaire d'indenter du Python est de n'utiliser que des espaces. La seconde méthode est de n'utiliser que des tabulations. Le code indenté avec un mélange de tabulations et d'espaces devrait être modifié pour n'utiliser que des espaces. Lors de l'appel de l'interpréteur Python en ligne de commande avec l'option `-t`, il générera des avertissements à propos de mélanges illégaux de tabulations et d'espaces. Lors de l'utilisation de `-tt`, ces avertissements deviennent des erreurs. Ces options sont vivement recommandées !

Pour les nouveaux projets, l'utilisation exclusive d'espace est vivement recommandée. Beaucoup d'éditeurs ont des fonctionnalités rendant ceci plus simple.

2.2 Longueur maximum d'une ligne

Limitez toutes les lignes à un maximum de 79 caractères.

Il y a encore de nombreux appareils qui sont limités à 80 caractère ; de plus, limiter les fenêtres à 80 caractères rend possible le fait d'avoir plusieurs fenêtres côte à côte. Le retour à la ligne par défaut de tels appareils nuit à la structure visuelle du code, le rendant plus difficile à comprendre. Veuillez donc à limiter toutes les lignes à un nombre maximum de 79 caractères. Pour de longs blocs de texte (docstrings ou commentaires), limiter la longueur à 72 caractères est recommandé.

La meilleure méthode "d'emballage" de longues lignes de code est d'utiliser la continuation implicite des lignes de Python, dans les parenthèses, crochets, et accolades. Si nécessaire, vous pouvez ajouter une paire supplémentaire de parenthèses autour de l'expression, mais utiliser un *backslash* \ est souvent plus esthétique. Soyez sûr(e) d'indenter proprement la ligne suivante. L'emplacement qui sera préféré pour un saut de ligne près d'un opérateur binaire est après l'opérateur, pas avant. Quelques exemples :

```
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)
```

2.3 Lignes vides

Séparer les fonctions les plus hautes par deux lignes vides.

Des lignes vides supplémentaires peuvent être utilisées (avec modération) pour séparer des groupes de fonctions. Les lignes vides peuvent être omises dans un ensemble de fonctions ne comprenant qu'une seule ligne (par exemple pour des implémentations abstraites).

Utilisez les lignes vides avec parcimonie dans les fonctions, pour indiquer des sections logiques.

Le code dans la distribution principale de Python 3.0 et les versions suivantes est l'UTF-8. Les fichiers utilisant l'UTF-8 doivent avoir une information sur l'encodage. L'UTF-8 n'est utilisé que dans les commentaires ou les docstrings qui ont une mention à un nom d'auteur qui requiert de l'UTF-8 ; sinon, utiliser `\x`, `\u`, ou `\U` pour échapper est la meilleur façon d'inclure des données non-ASCII dans les chaînes de caractères littérales. A partir de Python 3.4 et suivantes, l'encodage UTF-8 n'est plus à précisé et devient la valeur d'encodage par défaut pour tous les traitements (plus besoin des `\u` par exemple).

Pour Python 3.0 et les versions suivantes, la politique suivante est prescrite pour la bibliothèque standard : tous les identificateurs dans la bibliothèque standard Python DOIVENT être des identificateurs n'utilisant que l'ASCII, et DEVRAIENT, autant que possible, être des mots en Anglais (dans certains, des abréviations et des termes techniques qui ne sont pas en Anglais peuvent être utilisés). De plus, les chaînes littérales et les commentaires devraient également être en ASCII. Les seules exceptions sont :

- Des tests portant sur des fonctionnalités non-ASCII,
- Les noms d'auteurs. Les auteurs dont le nom n'est pas basé sur l'alphabet latin DOIVENT fournir une translittération de leur nom en alphabet latin.

Les projets open-source avec un public mondial sont encouragés à adopter cette politique

3 Imports

Les imports devraient être faits sur des lignes séparées, exemple :

```
# Oui :
import os
import sys
```

```
# Non :
import sys, os
```

Mais il est correct d'écrire ceci :

```
from subprocess import Popen, PIPE
```

Les imports sont toujours situés en haut du fichier, juste après d'éventuels commentaires et *docstrings* relatifs au module, et avant les globales et constantes du module.

Les imports devraient être groupés dans cette ordre :

1. imports de la bibliothèque standard
2. imports de modules tiers
3. imports spécifiques à l'application/bibliothèque locale

Vous pouvez mettre une ligne vide entre chaque groupe d'imports.

Les imports relatifs pour les imports intra-paquet sont hautement déconseillés. Utilisez toujours le nom absolu du paquet pour tous les imports. Les imports absolus sont plus portable et généralement plus lisibles.

4 Espaces blancs dans les expressions et les déclarations

4.1 Règles générales

4.1.1 Retirer les espaces supplémentaires dans les situations suivantes :

Immédiatement entre les parenthèses, accolades, ou crochets.

```
spam(ham[1], {eggs: 2}) # Oui
spam( ham[ 1 ], { eggs: 2 } ) # Non
```

Immédiatement avant une virgule, un point-virgule, ou un point.

```
if x == 4: print x, y; x, y = y, x # Oui
if x == 4 : print x , y ; x , y = y , x # Non
```

Immédiatement avant la parenthèse ouvrante qui commence la liste des arguments d'un appel de fonction :

```
spam(1) # Oui
spam (1) # Non
```

Immédiatement avant le crochet ouvrant qui commence un index ou une tranche :

```
dict['key'] = list[index] # Oui
dict ['key'] = list [index] # Non
```

Plus d'un espace autour d'un opérateur d'assignation (ou un autre) pour l'aligner avec un autre.

```
# Oui :
x = 1
y = 2
long_variable = 3
```

```
# Non :
x             = 1
y             = 2
long_variable = 3
```

4.1.2 Autres recommandations

Toujours entourer ces opérateurs binaires par un espace de chaque côté : assignation (`=`), assignation augmentée (`+=`, `-=`, etc), comparaisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), booléens (`and`, `or`, `not`).

4.1.3 Utiliser des espaces autour des opérateurs numériques :

```
# Oui :
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

```
# Non :
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

N'utilisez pas d'espaces autour du signe '=' quand il est utilisé pour indiquer un nom d'argument ou une valeur par défaut pour un paramètre.

```
# Oui :
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
# Non :
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

les déclarations combinées sont en général déconseillées.

```
  # Oui :
  if foo == 'blah':
      do_blah_thing()
  do_one()
  do_two()
  do_three()
```

```
  # Plutot que :
  if foo == 'blah': do_blah_thing()
  do_one(); do_two(); do_three()
```

Il est parfois bon de mettre un `if/for/while` avec un petit corps sur la même ligne, mais ne jamais faire ceci pour des déclarations multi-clauses. Évitez également les longues lignes.

```
# Deconseille :
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

```
# Surement pas :
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

5 Commentaires

Les commentaires qui contredisent le code sont pire que l'absence de commentaire. Ayez toujours pour priorité de garder les commentaires à jour quand le code change !

Le commentaires sont de préférence des phrases complètes. Si un commentaire est une expression ou une phrase, le premier mot doit être en majuscules, sauf s'il s'agit d'un identificateur qui commence par une minuscule (ne jamais modifier la casse d'identificateurs !)

Si un commentaire est court, la ponctuation finale peut être omise. Les commentaires multi-lignes consistent généralement en un ou plusieurs paragraphes construits de phrases complètes, chacune se terminant par une ponctuation.

Veuillez utiliser deux espaces après une ponctuation de fin de phrase.

Pour les programmeurs Python qui viennent de pays non-anglophones : veuillez toujours écrire vos commentaires en Anglais, à moins que vous ne soyez sûrs à 120% que le code ne sera jamais lu par quelqu'un qui ne parle pas votre langue.

5.1 Commentaires multi-lignes

Les commentaires multi-lignes s'applique généralement (voire tout le temps) au code qui les suit, et sont indentés sur le même niveau que ce code. chaque ligne du commentaire commence avec un `#` et un unique espace (à moins qu'il n'y ait du texte indenté dans le commentaire).

Les paragraphes à l'intérieur d'un commentaire multi-lignes sont séparés par une ligne contenant un unique `#`.

5.2 Commentaires en-ligne

Utilisez les commentaires en ligne avec modération.

Un commentaire en ligne est un commentaire qui est sur la même ligne qu'une déclaration. Les commentaires en-ligne doivent être séparés de la déclaration par au moins deux espaces. Ils doivent commencer par un `#` et un unique espace.

Les commentaires en-ligne ne sont pas nécessaire et ne servent qu'à distraire du plus important. Ne faites jamais ceci :

```
x = x + 1                 # Increment x
```

Mais plutôt :

```
x = x + 1                 # Compensate for border
```

5.3 Chaînes de documentation – Docstrings

Les conventions pour écrire de bonnes chaînes de documentation (aussi appelée "docstrings") sont immortalisées dans la PEP 257.

Une chaîne docstring est une chaîne de caractères qui apparaît en premier dans un module, une fonction, une classe ou dans la définition d'une méthode. Une telle chaîne devient l'attribut spécial `__doc__` de cet objet et est affiché avec la commande `help()`.

Il faut écrire des docstrings pour tous les modules publiques, fonctions, classes, et méthodes. Les docstrings ne sont pas nécessaire pour les méthodes non-publiques, mais vous pouvez avoir des commentaires décrivant ce que fait chacune d'elles. Ce commentaire doit être situé après la

ligne du "`def`".

Les commentaires situés ailleurs dans le code ne sont pas reconnus comme docstrings mais demeurent des commentaires (ils ne seront pas assignés à `__doc__`).

Pour garder une cohérence, utilisez toujours `"""` (triple double quotes/guillemets) autour des docstrings. Utilisez `r"""` si vous utilisez des backslashes dans votre docstring. Pour les doctrings utilisant des caractères unicodes, utilisez `u"""` pour les versions de python avant 3.4.

Il existe deux formats de docstrings :
— Sur une seule ligne,
— Multilignes.

5.3.1 Une seule ligne

Les docstrings sur une seule ligne doivent vraiment ne prendre qu'une seule ligne et doivent être utilisés pour des cas évidents. Par exemple :

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

Remarquez que les triples quotes sont utilisés même si le commentaire ne prend qu'une seule ligne.

Il n'y a pas de ligne vide avant ou après le docstring.

Le docstring est une phrase qui se termine par un point (ou plusieurs phrases). Il décrit les effets de la méthode ou de la fonction ("fais ceci, fait cela"), mais ce n'est pas une description. Par exemple n'écrivez jamais "retourne le pathname ...".

Le docstring d'une ligne ne doit pas être la signature rappelant les paramètres de la fonction/méthode. Ne pas faire ceci :

```
def function(a, b):
    """function(a, b) -> list"""
```

5.3.2 Multilignes

Les docstrings sur plusieurs lignes sont constituées d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module), suivie d'un saut de ligne, suivi d'une description plus longue. Respectez autant que faire se peut cette convention : une ligne de description brève, un saut de ligne puis une description plus longue.

La PEP 257 décrit les bonnes conventions pour écrire des docstrings. Veuillez noter que le `"""` qui fini une docstring multi-ligne est, de préférence, précédé par une ligne blanche. Par exemple :

```
    """Return a foobang

    Optional plotz says to frobnicate the bizbaz first.

    """
```

La docstring d'un module doit généralement dresser la liste des classes, exceptions et fonctions, ainsi que des autres objets exportés par ce module (une ligne de description par objet). Cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation. La documentation d'un package (la docstring se trouvant dans le fichier `__init__.py`) doit également dresser la liste des modules et sous-packages qu'il exporte.

La documentation d'une fonction ou méthode doit décrire son comportement et documenter ses arguments, sa valeur de retour, ses effets de bord, les exceptions qu'elle peut lever et les restrictions concernant son appel (quand ou dans quelles conditions appeler cette fonction). Les paramètres optionnels doivent également être documentés.

```
def complexe(reel=0.0, image=0.0):
    """Forme un nombre complexe.
    Parametres nommes :
    reel -- la partie reelle (0.0 par defaut)
    image -- la partie imaginaire (0.0 par defaut)
    """
    if image == 0.0 and reel == 0.0: return complexe_zero
    ...
```

6 Conventions de nommage

Les conventions de nommage de la bibliothèque Python sont un peu désordonnées, et nous ne pourrons jamais les rendre parfaitement uniformes ; cependant, voici les conventions de nommage actuellement recommandées. Les nouveaux modules et paquets (y compris les frameworks tiers) doivent être écrits avec ces standards, mais si une bibliothèque existante a un style différent, la cohérence interne est préférée.

6.1 Description des styles de noms

Il y a de nombreux styles de noms. Il est utile de pouvoir reconnaître lequel est utilisé, indépendamment de ce pour quoi il est utilisé.

On distingue les styles de noms suivants :
- `b` (lettre minuscule seule)
- `B` (lettre majuscule seule)
- `minuscules`
- `minuscules_avec_des_underscore`
- `MAJUSCULES`
- `MAJUSCULES_AVEC_DES_UNDERSCORE`
- `MotsEnMajuscules` (ou `MotsMaj`, ou `CamelCase`). Note : lors de l'utilisation d'abréviations, mettez en majuscules toutes les lettres de l'abréviation (par exemple : `HTTPServerError`)
- `casseMelangee` (diffère de `MotsEnMajuscules` par la minuscule initiale)

De plus, les formes spéciales suivantes avec des underscores préfixant/suffixant sont reconnu (ils peuvent généralement être combinés à toute convention de casse) :
- `_un_seul_underscore_prefixant` : faible indicateur d'"utilisation interne". Par exemple, "`from M import *`" n'importera pas les objets dont le nom commence par un underscore.
- `un_seul_underscore_suffixant_` : utilisé par "convention pour éviter les conflits avec des mots-clef Python, par exemple : `Tkinter.Toplevel(master, class_='ClassName')`
- `__doubles_underscore_prefixant_et_suffixant__` : objets "magique" ou attributs situés dans un espace de nom contrôlé par l'utilisateur. Exemple : `__init__`, `__import__`, ou `__file__` Ne jamais inventer de tels noms, n'utilisez que ceux documentés.

6.2 Conseils quant aux conventions de nommage

6.2.1 Noms à éviter

Ne jamais utilisez les caractères ‘l‘ (L minuscule), ‘O‘ (o majuscule), ou ‘I‘ (i majuscule) seuls comme nom de variables.

Dans certaines polices, ces caractères sont impossibles à distinguer des numériques un et zéro. Lorsque vous êtes tenté(e) d'utiliser ‘l‘, utilisez ‘L‘ à la place.

6.2.2 Noms de paquets et de modules

Les modules doivent avoir un court nom entièrement en minuscules. Les underscores peuvent être utilisés dans le nom du module si cela améliore la lisibilité. Les paquets Python doivent également avoir un court nom en minuscules, mais l'utilisation d'underscores est déconseillée.

Puisque que les noms de modules sont basés sur les noms de fichiers, et que certains systèmes de fichiers ne sont pas sensibles à la casse et tronquent les longs noms de fichiers, il est important que les noms de modules soient très courts ; ce n'est pas un problème sur les systèmes Unix, mais c'en est un quand le code est utilisé sur de vieilles version de Mac ou de Windows, ou sur DOS.

Quand un module d'extension en `C` ou en `C++` accompagne un module Python qui fourni une interface de plus haut niveau (plus orientée objet), le module `C/C++` utilise un underscore pour préfixe (par exemple : `_socket`).

6.2.3 Noms de variables globales

(En supposant que ces variables ne sont utilisées que pour un usage interne à un module.) Les conventions sont les même pour les fonctions.

6.2.4 Noms de fonctions

Les noms de fonctions doivent être en minuscules, les mots séparés par des underscores si nécessaire.

casseMelangee est également autorisée dans les cas où c'est déjà le style dominant (par exemple `threading.py`), pour garder la rétro-compatibilité.

Quand un argument de fonction est en conflit avec un mot-clef réservé, il est généralement bon d'ajouter un underscore à la fin de son nom, plutôt que d'utiliser une abréviation ou une modification de l'orthographe. Par exemple, "`print_`" est mieux que "`prnt`". (Le mieux reste peut-être d'éviter de tels conflits en utilisant un synonyme).

6.2.5 Noms de méthodes et variables d'instance

Utilisez les règles de nommage de fonction : minuscules avec mots séparés par des underscores si cela améliore la lisibilité.

Utilisez un unique underscore préfixant pour les méthodes et variables d'instance non-publiques.

6.2.6 Constantes

Les constantes sont généralement déclarées au niveau d'un module, et écrites en majuscules avec des underscores séparant les mots. Par exemple : `MAX_OVERFLOW` ou `TOTAL`.

7 Recommandations de programmation

7.1 Comparaisons

Les comparaisons avec des singletons comme `None` doivent toujours être faites avec '`is`' ou '`is not`', jamais avec les opérateurs d'égalité.

Aussi, évitez d'écrire "`if x`" quand vous voulez dire "`if x is not None`", par exemple pour tester si une variable ou un argument qui est pas défaut à `None` contient une autre valeur. L'autre valeur peut avoir un type (par exemple un conteneur) qui peut être `false` dans le cas d'un booléen !

7.2 Chaînes de caractères

7.2.1 Évitez le module string

Utilisez toujours les méthodes des chaînes de caractères plutôt que celles du module string.

Les méthodes des chaînes de caractères sont toujours plus rapides et ont la même API que les chaînes de caractères unicode. Contournez cette règle uniquement si vous avez besoin de compatibilité avec des versions de Python inférieures à 2.0.

7.2.2 Préfixe et suffixe

Utilisez de préférence `' '.startswith()` et `' '.endswith()` plutôt qu'une tranche pour tester un préfixe ou un suffixe, excepté si votre code doit fonctionner avec Python 1.5.2 (mais espérons que ce n'est pas le cas).

`startswith()` et `endswith()` sont plus propres, et sont moins enclins à des erreurs.

Exemple :

```
if foo.startswith('bar'): # Oui

if foo[:3] == 'bar': # Non
```

7.2.3 Comparaison de types

Les comparaisons de types doivent toujours utiliser `isinstance()` plutôt que de comparer directement les types.

```
if isinstance(obj, int): # Oui

if type(obj) is type(1): # Non
```

Lorsque vous vérifiez qu'un objet est une chaîne de caractères, gardez toujours à l'esprit que cela peut être une chaîne de caractères unicode ! Dans Python 2.3, `str` et `unicode` ont une classe de base commune, `basestring`, utilisez donc :

```
if isinstance(obj, basestring):
```

7.2.4 Séquences

Pour les séquences (chaînes de caractères, listes, tuples), utilisez le fait que des séquences vides valent False. Exemple :

```
# Oui :
if not seq:
if seq:
```

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

7.2.5 Booléens

Ne comparez pas des valeurs booléennes à `True` ou `False` avec `==`

```
if greeting: # Oui
if greeting == True: # Non
if greeting is True: # Encore pire
```

8 Génération de documentation

Les *docstrings* sont habituellement utilisés en interactif avec l'interpréteur pour connaître le rôle d'une fonction ou les paramètres qu'il prend.

Cependant, s'il faut fournir une documentation externe (page web par exemple) sur l'utilisation ou la description des fonctions ou modules, il sera dommage ne pas pouvoir tirer partie des docstrings. Heureusement il existe plusieurs applications (Python ou non) qui peuvent analyser le code (et surtout les docstrings) afin de générer automatiquement cette documentation externe. Par exemple, il existe *epydoc* et *Sphinx*.

Afin d'enrichir un peu plus la documentation générée, ces analyseurs de docstring permettent d'introduire des mots clés dans les commentaires pour une documentation plus précise. Les conventions peuvent être celles définies par Epytext Markup Language, ou reStructuredText. Les deux syntaxes sont assez similaires.

Epytext	reStructuredText
`@tag: texte ...`	`:tag: texte ...`
`@tag arg: texte ...`	`:tag arg: texte ...`

Dans cette section seront décrits les mots clés relatifs à Epytext/epydoc mais les mêmes mots clés sont présents dans le format *rst*.

8.1 Paramètres de fonctions et de méthodes

Ces mots clés de description doivent être placés dans le *docstring* associé à la fonction ou à la méthode.

`@param p: ...` La description du paramètre `p` pour la fonction ou la méthode.

`@type p: ...` Le type attendu pour le paramètre `p`.

`@return: ...` La valeur de retour pour de la fonction ou méthode.

`@rtype: ...` Le type de la valeur de retour.

`@keyword p: ...` La description du mot clé paramètre `p`.

`@raise e: ...` La description des circonstances pour lesquelles la fonction ou la méthode lève une exception `e`.

le champs `@param` doit être utilisé pour documenter tous les paramètres explicites, le champs `@keyword` doit être utilisé seulement pour les mots clés des paramètres non explicites.

```
def plant(seed, *tools, **options):
    """
    @param seed: The seed that should be planted.
    @param tools: Tools that should be used to plant the seed.
    @param options: Any extra options for the planting.

    @keyword dig_deep: Plant the seed deep under ground.
    @keyword soak: Soak the seed before planting it.
    """
    #[...]
```

Comme le champs @type accepte un texte arbitraire, il n'y a pas création automatique d'un lien de référence croisée vers le type spécifié, et le texte n'est pas écrit par défaut avec une police à largeur fixe. Si vous voulez créer un lien de référence croisée vers le type, ou écrire en police à largeur fixe, dans ce cas vous devez utiliser les balises adéquates :

```
def ponder(person, time):
    """
    @param person: Who should think.
    @type person: L{Person} or L{Animal}
    @param time: How long they should think.
    @type time: C{int} or C{float}
    """
    #[...]
```

8.2 Variables

Ces champs sont généralement placés dans le *docstring* d'un module ou d'une classe. Si les sources sont également accessibles, il est possible de documenter les variables directement dans leur propre *docstring*.

`@var v: ...` description d'une variable de module `v`.

`@type v: ...` le type de la variable `v`.

`@ivar v: ...` description d'une instance de classe `v`.

`@cvar v: ...` description d'une variable de classe `v` statique.

Epydoc considère comme variables de classe celles qui sont directement définies dans le corps de la classe.

Python ne supporte pas directement les *docstrings* de variables : il n'y a pas d'attribut qui peut être attaché aux variables et récupéré comme par exemple avec l'attribut `__doc__` pour les modules pour les fonctions.

Bien que le langage ne le permette pas directement, Epydoc supporte les docstrings de variables : si une affectation de variable est directement suivie par une chaîne de caractères de commentaire avec `"""`, alors cette cette affectation est traitée comme un docstring pour cette variable.

Les variables peut également être documentée en utilisant un commentaire simple. Si l'affectation d'une variable est immédiatement précédée par une simple ligne de commentaire commençant par `#:`, ou si elle est suivie sur la même ligne par le même type de commentaire, alors il est traité comme un docstring pour cette variable :

```
x = 22
"""Docstring for x"""

#: docstring for x
x = 22

x = 22 #: docstring for x
```

Remarquez bien que le docstring de variable n'existe que si les sources sont accessible car il n'est pas possible récupéré à cette donnée à partir de la variable dans l'interpréteur.

8.3 Notes et Warnings

Il est possible d'introduire des commentaires indiquant une remarque ou des commentaires sur le statut du code *via* les champs suivant :

`@note: ...` Une note à propos de quelque chose (module, méthode, etc). Plusieurs champs *note* peuvent être utilisés.

`@attention: ...` Une note importante.

`@bug: ...` La description d'un bug dans un objet (module, méthode, etc). Plusieurs champs bug peuvent être utilisés pour indiquer des bugs séparés.

`@version: ...` La version courante de l'objet (module, méthode, etc).

`@todo [ver]: ...` Un changement planifié pour un objet. Si l'argument optionnel `ver` est indiqué, cela indique dans quelle version ce changement prendra effet. Plusieurs champs `todo` peuvent être utilisés pour indiquer de multiples changements prévus.

`warning: ...` Une note très importante à laquelle il faut faire attention. Plusieurs champs `warning` peuvent être utilisés séparément.

8.4 Informations bibliographiques

Généralement, en début de module principale ou de tous les modules, les auteurs placent des informations bibliographique sur le nom des auteurs, la licence, etc. Des champs sont ainsi également définis pour générer la documentation. Toutefois, le système de génération est capable d'extraire ces informations des méta-données du programme python (cf. section 8.7).

`@author: ...` Le(s) auteur(s) de l'objet (module). Plusieurs champs `author` peuvent être utilisés dans le cas de plusieurs auteurs.

`@organization: ...` L'organisation ou entreprise qui a créé ou qui maintient l'objet.

`@copyright: ...` L'information de copyright.

`@license: ...` L'information de licence.

`@contact: ...` Les informations afin de pouvoir contacter les auteurs ou ceux qui maintiennent le module, la classe, la fonction ou la méthode. Il est possible d'utiliser plusieurs champs `contact` pour indiquer plusieurs personnes à contacter.

8.5 Autres champs

`@summary: ...` Un résumé décrivant l'objet. Cette description inhibe le résumé par défaut qui est construit à partir de la première phrase de la description de l'objet.

`@see: ...` La description d'un sujet au sujet. `see` est généralement utilisé pour indiquer des liens de référence croisées ou des hyperliens externes qui se rapportent au sujet.

8.6 Champs synonymes

Plusieurs champs possèdent des synonymes ou des mots clés alternatifs. La liste suivante en reprend quelques uns, les plus communs.

Nom	Synonyme(s)
`@param p: ...`	`@arg p: ...`
`@keyword p: ...`	`@kwarg p: ...` `@kwparam p: ...`
`@warning: ...`	`@warn: ...`

8.7 Variables de meta-donnée

Plusieurs variables de module sont habituellement utilisées comme méta-données du programme. Epydoc peut utiliser les informations contenues dans ces méta-données comme champs alternatifs pour générer la description du module.

Le tableau suivant liste les noms de variables et les champs associés.

Tag	Variable
`@author`	`__author__`
`@authors`	`__authors__`
`@contact`	`__contact__`
`@copyright`	`__copyright__`
`@license`	`__license__`
`@date`	`__date__`
`@version`	`__version__`