ENTWANNE

Zeste de savoir

Les secrets d'un code pythonique

12 août 2019

Table des matières

1. Zen of Python . . . 2
   - 1.1. Beautiful is better than ugly. . . . 2
   - 1.2. Explicit is better than implicit. . . . 2
   - 1.3. Simple is better than complex. . . . 3
   - 1.4. Complex is better than complicated. . . . 3
   - 1.5. Flat is better than nested. . . . 3
   - 1.6. Sparse is better than dense. . . . 4
   - 1.7. Readability counts. . . . 4
   - 1.8. Special cases aren't special enough to break the rules. . . . 4
   - 1.9. Although practicality beats purity. . . . 4
   - 1.10. Errors should never pass silently. . . . 5
   - 1.11. Unless explicitly silenced. . . . 5
   - 1.12. In the face of ambiguity, refuse the temptation to guess. . . . 5
   - 1.13. There should be one – and preferably only one – obvious way to do it. . . . 6
   - 1.14. Although that way may not be obvious at first unless you're Dutch. . . . 6
   - 1.15. Now is better than never. . . . 6
   - 1.16. Although never is often better than *right* now. . . . 7
   - 1.17. If the implementation is hard to explain, it's a bad idea. . . . 7
   - 1.18. If the implementation is easy to explain, it may be a good idea. . . . 7
   - 1.19. Namespaces are one honking great idea – let's do more of those ! . . . 7
   - 1.20. Fin ? . . . 8
2. Les règles de style . . . 8
3. Les autres principes . . . 9
   - 3.1. Keep it simple, stupid (*KISS*) . . . 9
   - 3.2. Don't repeat yourself (*DRY*) . . . 9
   - 3.3. You ain't gonna need it (*YAGNI*) . . . 10
   - 3.4. We're all consenting adults here . . . 10
   - 3.5. *Easier to ask forgiveness than permission* (*EAFP*) . . . 11
4. Les mécanismes du langage . . . 11
   - 4.1. *Unpacking* . . . 11
   - 4.2. Conditions . . . 12
   - 4.3. Boucle `for` . . . 13
   - 4.4. Listes en intension . . . 13
   - 4.5. Générateurs . . . 14
   - 4.6. Exceptions . . . 14
   - 4.7. Décorateurs . . . 14
   - 4.8. Gestionnaires de contextes . . . 15
5. La bibliothèque standard . . . 15
   - 5.1. *Built-in* . . . 15
   - 5.2. Autres modules . . . 16

6. Les bons réflexes 17

Avez-vous déjà vu… un code pythonique ?

« Pythonique », c'est un terme que l'on rencontre souvent au sein d'articles ou sur des forums, pour qualifier un code Python bien conçu, un code idiomatique (en accord avec les règles d'usage du langage, et donc compréhensible par tout développeur).

Seulement, la distinction entre un bon code et un autre peut s'avérer floue, cet article a justement pour but de détailler les règles qui font qualifier un code de pythonique ou non.

1. Zen of Python

Tout commence par la fameuse PEP20 , *The Zen of Python*, écrite par Tim Peters, et qui exprime les valeurs du langage.

Celle-ci énonce les règles suivant un poème. On peut la retrouver *via* l'instruction `import this` dans un interpréteur Python.

1.1. Beautiful is better than ugly.

Le beau est préférable au laid.

Le point de départ. Un bon code Python se doit d'être beau, c'est à dire agréable à regarder. Nous reviendrons par la suite sur les règles de style, qui définissent plus précisément en quoi consiste un beau code.

Cela se voit en Python par l'utilisation de mots plutôt que de symboles pour certains opérateurs (`and`, `or`, `not`, `in`), qui rendent les expressions plus proches du langage naturel.

```
1 if number > 0 and number not in invalid_numbers:
2     ...
```

1.2. Explicit is better than implicit.

L'explicite est préférable à l'implicite.

Un développeur doit pouvoir lire un code Python sans se demander sans cesse ce que fait telle ou telle ligne. Utiliser des noms et des constructions explicites permet de limiter ce genre de problèmes.

Par exemple, en programmation objet, lors d'un héritage et de la surcharge de la méthode d'initialisation (`__init__`), il convient d'appeler explicitement la méthode de la classe parente. Cela ne sera jamais fait automatiquement dans le dos du développeur, afin d'avoir la main sur le comportement voulu.

```python
class User:
    def __init__(self, name):
        self.name = name

class SecureUser(User):
    def __init__(self, name, password):
        super().__init__(name)
        self.password = password
```

1.3. Simple is better than complex.

Le simple est préférable au complexe.

Certaines structures du langage vont s'avérer plus complexes que d'autres. L'application d'un décorateur est une instruction complexe, par les mécanismes qu'elle met en œuvre : patron de conception décorateur, fonctions passées implicitement en paramètre.

```python
@staticmethod
def method():
    ...
```

1.4. Complex is better than complicated.

Le complexe est préférable au compliqué.

Il convient déjà de bien faire la différence entre les deux termes.

« compliqué » se rapporte à l'utilisation. Un code compliqué est difficile à relire et à maintenir. Un code complexe utilise des mécanismes avancés, mais il peut être simple à appréhender. Pour reprendre l'exemple précédent, l'application d'un décorateur n'est pas compliquée.

1.5. Flat is better than nested.

Le plat est préférable à l'imbriqué.

Quand on lit un code, il est facile de perdre le fil et d'oublier à quel endroit on se trouve. D'autant plus si de nombreux niveaux d'imbrications se succèdent.

Pour palier à ce problème, on préférera produire du code plat chaque fois que cela est possible, et ainsi éviter les imbrications inutile. Dans le cadre d'une fonction, on choisira par exemple de retourner directement quand des préconditions ne sont pas validées, plutôt que de placer le contenu de notre fonction dans plusieurs sous-niveaux de conditions.

```python
def print_items(obj):
    if not hasattr(obj, 'items'):
        return
    for item in obj.items:
        print(item)
```

1.6. Sparse is better than dense.

L'aéré est préférable au dense.

Un code compréhensible est un code aéré. La syntaxe même du langage se base sur l'indentation pour séparer les blocs logiques. L'aération du code y est donc une valeur très importante.

1.7. Readability counts.

La lisibilité compte.

Vous, et les autres développeurs du projet, passerez probablement plus de temps à lire votre code qu'à l'écrire. Le code se doit donc d'être lisible facilement, pour ne pas faire perdre de temps à tous.

La lisibilité passera par de nombreux points évoqués par les autres directives, mais aussi par un choix judicieux des noms de fonctions et variables par exemple.

```python
def reset_password(*users, password=''):
    for user in users:
        user.password = password
```

1.8. Special cases aren't special enough to break the rules.

Les cas spéciaux ne le sont pas assez pour briser les règles.

Ce principe est celui de la cohérence. Les mêmes règles s'appliquent pour tous, ce n'est pas parce qu'un bout de code semble sortir du lot qu'il y déroge. Même un code imbriqué doit rester lisible, par exemple.

1.9. Although practicality beats purity.

Bien que la praticité prévale sur la pureté.

Et cette règle, qui nuance la précédente, représente le bon sens. Il peut devenir nécessaire d'outrepasser les règles pour des raisons pratiques, telles que des questions de performances. Cela doit dans tous les cas rester anecdotique.

1.10. Errors should never pass silently.

Les erreurs ne devraient jamais se produire silencieusement.

Quand une erreur se produit c'est qu'il y a un problème, quel qu'il soit. Ce problème ne doit jamais être masqué au développeur.

```
>>> def division(a, b):
...     return a / b
...
>>> division(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in division
ZeroDivisionError: division by zero
```

1.11. Unless explicitly silenced.

À moins d'être explicitement tues.

Le développeur peut ensuite choisir d'ignorer une erreur en particulier, car celle-ci est attendue dans ce cas précis. Mais il le fera de façon explicite, avec un bloc `try`/`except` par exemple.

```
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return float('nan')
```

1.12. In the face of ambiguity, refuse the temptation to guess.

En cas d'ambiguïté, résister à la tentation de deviner.

Deviner implique un choix, choix qui ne sera pas forcément clair pour tous les développeurs. S'il n'est pas clair, c'est qu'il n'est pas explicite.

Par exemple, dans le cas d'une addition entre de valeurs de types `str` et `int`, il y a ambiguïté entre le fait de choisir de convertir les deux opérandes en nombres ou en chaînes de caractères. Aucune conversion implicite ne sera effectuée, il faudra convertir manuellement les deux opérandes en types compatibles.

```
>>> a = '5'
>>> a + 1
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> int(a) + 1
6
>>> a + str(1)
'51'
```

1.13. There should be one – and preferably only one – obvious way to do it.

Il devrait y avoir une – et de préférence une seule – manière évidente de le faire.

Python prône le fait qu'il existe toujours une manière optimale de procéder, et donc que toutes ne se valent pas. Celle-ci est préférable car évidente.

Nous y reviendrons plus loin avec les mécanismes du langage, mais la manière évidente d'itérer sur des nombres est par exemple d'utiliser une boucle `for`.

```
for i in range(100):
    print(apply_func(i))
```

1.14. Although that way may not be obvious at first unless you're Dutch.

Bien que cette manière ne vous semble pas évidente au premier abord, à moins que vous ne soyez néerlandais.

Cependant, l'évidence n'est pas innée. Elle vient avec la pratique, et est entre autres dictée par cette *PEP*. La manière évidente est la manière la plus idiomatique.

Cette règle se termine par une note humoristique sur Guido van Rossum, néerlandais, le créateur de Python.

1.15. Now is better than never.

Maintenant est préférable à jamais.

La procrastination est l'ennemie du développeur Python. Si vous avez besoin d'une fonctionnalité manquante, implémentez-la, ne codez pas de rustines temporaires pour y revenir « plus tard ».

1.16. Although never is often better than *right* now.

Bien que jamais soit souvent préférable à tout de suite.

Assurez-vous cependant que cette fonctionnalité soit vraiment nécessaire. Dans le cas contraire, il peut être préférable de ne pas perdre trop de temps dessus pour le moment.

Ce point sera détaillé par la suite quand nous aborderons le principe *YAGNI*.

1.17. If the implementation is hard to explain, it's a bad idea.

Si l'implémentation est difficile à expliquer, c'est une mauvaise idée.

Une implémentation difficile à expliquer est compliquée, elle produira du code compliqué. On préférera donc l'éviter au profit d'une implémentation plus simple, plus facile à expliquer.

1.18. If the implementation is easy to explain, it may be a good idea.

Si l'implémentation est facile à expliquer, il peut s'agir d'une bonne idée.

Pour autant, être facile à expliquer n'en fait pas une bonne implémentation. C'est une bonne chose, mais ce n'est pas un critère suffisant.

Il est facile d'expliquer comment concaténer plusieurs chaînes de caractères : on itère sur notre ensemble de chaînes, et on les concatène chacune à une chaîne finale à l'aide de l'opérateur `+`. Pourtant, celle solution est à proscrire, dû à l'inefficacité de l'opérateur de concaténation, et à la présence d'une méthode `join` bien plus lisible.

```
>>> tags = ['<html>', '<body>', '<p>', 'text', '</p>', '</body>',
    '</html>']
>>> ''.join(tags)
'<html><body><p>text</p></body></html>'
```

1.19. Namespaces are one honking great idea – let's do more of those !

Les espaces de noms sont une sacrée bonne idée – utilisons-les plus souvent !

Les espaces de noms sont créés à l'aide des paquets, modules et objets, ils permettent de diviser les classes et fonctions en ensembles logiques. Ils évitent aussi les conflits de noms, un même nom pouvant être utilisé pour des valeurs différentes dans des espaces différents. On aimera alors en user pour bien classifier nos objets.

```
>>> import math, cmath
>>> math.exp(0)
1.0
```

```
>>> cmath.exp(0)
(1+0j)
```

1.20. Fin ?

Tim Peters avait initialement annoncé que son *Zen of Python* contiendrait 20 directives. Si vous y avez prêté attention, on en compte plutôt 19. Où est passée cette fameuse 20ème règle ?

Certains évoquent qu'elle pourrait être implicite, une simple ligne vide. Une ligne vide qui rappellerait l'aération.

2. Les règles de style

Les règles de style permettent d'assurer une certaine lisibilité d'un code, elles sont un socle commun à tous les projets Python. Ces règles sont énoncées par la PEP8 .

Le premier principe à respecter est la cohérence. Il se peut que vous ayez affaire à une bibliothèque ne respectant pas la PEP8. Dans ce cas, adaptez-vous au style de cette bibliothèque. La concordance avec les règles de style générales passe en second plan.

Aussi, la lisibilité prévaut sur tout le reste. Si, dans votre cas précis, une règle nuit à la compréhension d'une ligne, ne l'appliquez pas. *Practicability beats purity.*

Je ne vais pas détailler ici l'ensemble des directives, je vous laisse consulter la PEP pour cela. Retenez qu'elle concerne l'indentation et l'aération, les imports, les commentaires, les conventions de nommage, et d'autres recommandations plus générales.

Les commentaires sont très importants pour la compréhension du code. Ils expliquent comment fonctionne le code et pourquoi il est implémenté de telle manière. Ils sont complétés par les *docstrings*, des chaînes de caractères en en-tête des modules, classes et fonctions qui permettent de les documenter (d'expliquer comment s'utilise le code). Les *docstrings* d'un objet Python sont accessibles *via* la fonction `help` appelée sur cet objet. On y retrouvera d'autres informations telles que les annotations (spécifications des types des paramètres et du type de retour des fonctions).

```
>>> def addition(a : int, b : int) -> int:
...     "Return the sum of numbers `a` and `b`."
...     return a + b
...
>>> help(addition)
```

Sachez aussi que des outils sont à votre disposition pour analyser le style de votre code et son respect des conventions. Ils sont divers et variés, mais les deux plus connus sont probablement `pylint` et `flake8`.

3. Les autres principes

La programmation Python repose sur d'autres principes généraux, décrits dans cette section.

3.1. Keep it simple, stupid (*KISS*)

Garde ça simple, stupide

Ce premier principe se rapproche clairement du *Simple is better than complex*. Le code doit toujours rester le plus simple possible, afin de rester lisible pour les autres contributeurs.

Cela s'illustre par la syntaxe même du langage, qui comprend peu de constructions différentes, mais doit aussi se retrouver dans le code produit. Les fonctions, par exemple, doivent être dédiées à une unique fonctionnalité, de même pour les classes et leurs méthodes.

En parlant de classes, il est inutile de créer de nouvelles classes trop vite, là où les types primitifs du langage pourraient répondre au besoin. Par exemple, pour un objet qui ne contiendrait que des données, associées à aucune méthode, un dictionnaire fait très bien l'affaire.

```
user = {'username': 'guido', 'realname': 'Guido van Rossum',
    'password': '12345'}
```

Le principe s'exprime aussi par le fait de ne pas créer de hiérarchie de classes trop complexe, et même d'ailleurs de ne pas user d'héritage quand ce n'est pas nécessaire (penser au *duck-typing*). De même, Python dispose d'outils puissants (décorateurs, générateurs, métaclasses), qui doivent être utilisés judicieusement, quand ils ne nuisent pas à la simplicité.

3.2. Don't repeat yourself (*DRY*)

Ne te répète pas

Cette seconde règle a pour but d'éviter la redondance. Le code dupliqué est plus difficile à maintenir, car chaque modification doit être répercutée sur toutes les occurrences du code.

La répétition peut se comprendre à petite échelle : par exemple une même ligne répétée à deux endroits du code. Au-delà, une factorisation est nécessaire, afin de dédier une fonction à ce comportement.

```
import sys
import random

def errlog(template, *args):
    print(template.format(*args), file=sys.stderr)

secret = random.randint(0, 100)
guess = int(input('Entrez un nombre entre 0 et 100: '))
```

```
 9
10 if guess < secret:
11     errlog('Nombre {} trop petit', guess)
12 elif guess > secret:
13     errlog('Nombre {} trop grand', guess)
14 ...
```

Notre fonction `errlog` permet ici de factoriser le formatage et l'affichage de messages d'erreur.

3.3. You ain't gonna need it (*YAGNI*)

Tu n'en auras pas besoin

Ce principe est plus une ligne de conduite pour le processus de développement. Il est inutile de développer maintenant une fonctionnalité qui ne servira peut-être jamais. Il est préférable de s'attaquer d'abord à ce qui est actuellement nécessaire.

Développer une fonctionnalité trop tôt présente de plus d'autres problèmes :

— Inutilisée, elle restera inconnue des autres développeurs ;
— Si elle vient à être utilisée, elle ne le sera peut-être pas dans les termes actuellement définis ;
— La fonctionnalité devra être continuellement testée tout le long du développement du projet, et potentiellement déboguée ;
— Enfin, elle pourrait entrer en conflit avec d'autres fonctionnalités requises.

3.4. We're all consenting adults here

Nous sommes ici entre adultes consentants

Ou plus clairement, les développeurs sont conscients et responsables de leurs actes. Cela s'illustre par la manière de protéger des attributs en Python, en les préfixant par un `_`.

En soi, rien n'empêche d'accéder depuis l'extérieur à un tel attribut. Mais le préfixe signale au développeur qu'il accède à un état interne, que sa modification pourrait compromettre le comportement normal de l'objet, et qu'il le fait donc en connaissance de cause.

```
1 class MyObject:
2     def __init__(self):
3         self._internal = 'internal state'
4
5 obj = MyObject()
6 print(obj._internal)
```

En parlant d'attributs, on préférera toujours en Python un accès direct aux attributs plutôt que des méthodes *getter/setter*. Quitte à passer par des propriétés s'il est nécessaire que la récupération ou la modification de l'attribut soit dynamique.

3.5. Easier to ask forgiveness than permission (EAFP)

Il est plus facile de demander pardon que la permission

Python fait partie des langages qui considèrent qu'il est plus simple d'essayer puis de gérer les erreurs, que de demander la permission en amont.

Pour gérer l'ouverture d'un fichier, par exemple, on préférera faire appel à `open`, et traiter les différentes exceptions qui pourraient se produire (fichier inexistant, droits insuffisants, etc.), plutôt que de tester une à une ces différentes conditions.

```
try:
    with open('filename', 'r') as f:
        handle_file(f)
except FileNotFoundError as e:
    errlog('Fichier {!r} non trouvé', e.filename)
except PermissionError as e:
    errlog('Fichier {!r} non lisible', e.filename)
```

Cette manière de procéder a aussi l'avantage d'être plus sûre en Python. En effet, dans le cas où l'on testerait d'abord l'existence du fichier, rien ne nous garantit qu'il serait toujours présent au moment de l'ouverture proprement dite (il peut être supprimé par un autre programme entretemps).

Ce principe s'oppose au *LBYL* (*Look before you leap*, *Regarde avant d'essayer*), préconisé par d'autres langages comme le C.

4. Les mécanismes du langage

On reconnaît généralement un bon code Python à l'utilisation des mécanismes qui lui sont propres.

4.1. Unpacking

Un premier point à aborder est celui de l'*unpacking* (ou déconstruction), une technique qui permet l'assignation de plusieurs variables en une seule instruction.

Vous l'avez probablement déjà rencontré comme exemple pour échanger les valeurs de deux variables.

```
>>> a = 5
>>> b = 2
>>> a, b = b, a
>>> print(a, b)
2 5
```

Ce qui se passe en interne lors de la 3ème ligne est la création d'un *tuple* `(b, a)`, qui est ensuite déconstruit et son contenu stocké dans les variables `a` et `b`.

Mais l'*unpacking* ne se limite pas à cela, et permet aussi de déconstruire des structures imbriquées (*tuples*, listes, chaînes de caractères, dictionnaires).

```
>>> l = [0, (1, 2, {3: 'foo', 4: 'bar'}), 5]
>>> a, (b, c, (d, e)), f = l
>>> print(a, b, c, d, e, f)
0 1 2 3 4 5
>>> x, y, z = 'bar'
>>> print(x, y, z)
b a r
```

L'*unpacking* est une manière élégante de séparer les éléments d'une liste, il est donc courant de l'employer en Python.

Nous n'aborderons pas ici les constructions plus complexes de l'*unpacking*, rendue possible grâce à l'opérateur *splat*, comme décrit ici .

4.2. Conditions

Toute valeur en Python peut s'évaluer sous forme d'un booléen, il n'est donc pas nécessaire de la convertir préalablement. Les valeurs `None`, `0` et le conteneurs vides (`''`, `()`, `[]`, `set()`, etc.) s'évaluent à `False`. Les autres nombres, les conteneurs non vides, et plus généralement toute valeur qui n'est pas explicitement fausse s'évaluent à `True`.

Ainsi, pour tester si une chaîne `s` n'est pas vide, il suffit de faire une condition sur `s`. On ne convertira jamais la valeur en booléen pour la comparer à `True` ou `False`.

```
if s:
    print("s n'est pas vide")
```

L'usage de ternaires est aussi à privilégier quand on souhaite évaluer des expressions conditionnelles courtes.

```
name = user.name if user is not None else 'anonymous'
```

On notera l'utilisation de l'opérateur `is` pour la comparaison avec `None`. Ce dernier étant une constante unique, `is` permet d'en assurer la singularité.

4.3. Boucle `for`

Un mécanisme important du langage est le protocole d'itération, mis en œuvre par la boucle `for`.

En Python, la boucle `for` doit toujours être privilégiée pour itérer sur un ensemble d'éléments. Si vous recourrez à une boucle `while` pour itérer, c'est probablement que vous avez un problème de conception ou méconnaissez les fonctions qui pourraient vous être utiles. Cet ensemble d'éléments ne prend pas toujours la forme d'une liste, il peut s'agir d'un dictionnaire, d'un fichier, d'un intervalle de nombres (`range`).

Et ceci est valable pour toutes les variables qui devraient prendre des valeurs successives à chaque itération. Ainsi, on s'orientera vers `zip` pour itérer sur plusieurs éléments à la fois, vers `enumerate` pour itérer en gardant trace de l'index dans la liste, ou encore vers des constructions plus complexes du module `itertools` que nous verrons plus loin.

```
names = ['Alex', 'Alice', 'Bob']
ages = [45, 27, 74]

for name, age in zip(names, ages):
    print(name, age)

for i, (name, age) in enumerate(zip(names, ages)):
    print(i, name, age)
```

Nous retrouvons dans cette construction l'*unpacking* abordé plus haut, qui peut donc s'utiliser aussi pour les boucles `for`.

4.4. Listes en intension

Outre la boucle `for`, le protocole d'itération est aussi représenté par les listes en intension, qui doivent être utilisées dès que possible, tant qu'elles ne nuisent pas à la lisibilité bien sûr.

Pour construire la liste des carrés des nombres de 0 à 9, on utilisera par exemple le code suivant, plutôt qu'une boucle multi-lignes et un remplissage de liste manuel.

```
squares = [i**2 for i in range(10)]
```

On retrouve la même construction pour les dictionnaires en intension.

```
squares_set = {i**2 for i in range(10)}
squares_dict = {i: i**2 for i in range(10)}
```

4.5. Générateurs

Un autre mécanisme est celui des générateurs (et des générateurs en intension), à utiliser quand il n'est pas nécessaire d'avoir une représentation complète d'un ensemble en mémoire. Si notre liste `squares` a simplement pour but de calculer la somme des éléments (`sum(squares)`), nous lui préférerons la version utilisant un générateur, évitant ainsi le stockage inutile de la liste.

```
sum_squares = sum(x**2 for x in range(10))
```

4.6. Exceptions

La gestion d'erreurs est réalisée en Python à l'aide d'un mécanisme d'exceptions, mais les exceptions ne se limitent pas à cela. Le protocole d'itération décrit plus haut s'appuie par exemple sur une exception `StopIteration` levée en fin de boucle.

Vos traitements défectueux doivent toujours remonter une exception adaptée au problème, et décrivant au mieux sa raison. Les types d'exceptions sont généralement hiérarchisés de façon à représenter le problème à différents niveaux d'abstractions.

Si vous êtes par exemple amené à développer une bibliothèque, il est courant que toutes ses exceptions héritent d'une même base permettant facilement d'attraper toutes les erreurs de la bibliothèque. Dans le cas d'un champ manquant lors de l'analyse du fichier de configuration d'un composant de votre bibliothèque `mylib`, vous pourriez avoir une exception de type `mylib.FieldMissingError` héritant de `mylib.ParseError` et elle même de `mylib.Error`.

De l'autre côté, il est conseillé d'attraper judicieusement les exceptions. Si vous souhaitez traiter un tel problème de champ manquant, vous attraperez l'exception `mylib.FieldMissingError` plutôt que `mylib.Error` qui serait ici trop générale.

4.7. Décorateurs

Les décorateurs, utilisés à bon escient, sont aussi une particularité du langage. On reconnaît un code idiomatique à l'utilisation des décorateurs de la bibliothèque standard (`staticmethod`, `classmethod`, `property`).

```
class Circle:
    def __init__(self, cx, cy, radius):
        self.cx, self.cy = cx, cy
        self.radius = radius

    @classmethod
    def from_diameter(cls, ax, ay, bx, by):
        cx, cy = (ax + bx) / 2, (ay + by) / 2
        diam = ((ax - bx)**2 + (ay - by)**2)**0.5
```

```
10          return cls(cx, cy, diam / 2)
11
12      @property
13      def area(self):
14          return math.pi * self.radius**2
```

La définition de ses propres décorateurs ne doit en revanche avoir lieu que si elle permet un gain net en lisibilité par rapport aux autres solutions envisagées.

4.8. Gestionnaires de contextes

Enfin, je voudrais aborder les gestionnaires de contexte (`with`), et notamment l'ouverture de fichiers, qui doit toujours passer par l'utilisation d'un bloc `with`. Ce bloc permet en effet d'automatiser des opérations de libération des ressources, et ce en tous les cas (déroulement normal ou erreur).

```
1 with open('hello.txt', 'w') as hello_file:
2     print('Hello World!', file=hello_file)
```

5. La bibliothèque standard

Un code pythonique se doit d'exploiter au mieux les modules de la bibliothèque standard. Il convient alors de les connaître dans les grandes lignes, pour être en mesure de les utiliser quand le besoin se fait sentir.

5.1. *Built-in*

Cela commence par la bonne utilisation des fonctions *built-in* . Savoir comment elles s'utilisent, connaître leurs paramètres (notamment le paramètre `key` des fonctions `min`, `max` et `sorted`).

Les types *built-in* sont aussi à regarder, afin d'en connaître les principales méthodes. Je pense par exemple à la méthode `format` des chaînes de caractères qui permet facilement de composer plusieurs valeurs en une chaîne.

```
1 print('{} + {} = {}'.format(2, 3, 2 + 3))
```

Je voudrais aussi aborder les méthodes `get` et `setdefault` des dictionnaires, qui permettent de gérer facilement les éléments manquants.

```
>>> database = {'foo': 123}
>>> database.get('bar')
>>> database.get('bar', 0)
0
>>> database.setdefault('letters', []).append('a')
>>> database.setdefault('letters', []).append('b')
>>> database
{'foo': 123, 'letters': ['a', 'b']}
```

Ou encore le constructeur des conteneurs standards (`list`, `tuple`, `dict`, `set`), qui accepte un autre itérable en paramètre.

```
>>> names = ['Alex', 'Alice', 'Bob']
>>> ages = [45, 27, 74]
>>> list(enumerate(names))
[(0, 'Alex'), (1, 'Alice'), (2, 'Bob')]
>>> dict(zip(names, ages))
{'Alex': 45, 'Alice': 27, 'Bob': 74}
```

Nous retrouvons enfin les exceptions *built-in* et leur hiérarchie.

On distinguera par exemple les `TypeError` pour relever une erreur due au type d'une variable, et les `ValueError` quand la valeur est du bon type mais ne correspond pas à ce qui est attendu. On notera aussi `IndexError` et `KeyError`, respectivement pour un index ou une clef non trouvée dans un conteneur.

5.2. Autres modules

Le module `collections` comporte d'autres structures de données essentielles au langage : `OrderedDict`, `namedtuple`, `Counter`, ou encore `defaultdict` qui sera préférable à une utilisation systématique de `setdefault`. Des développeurs débutants auront le réflexe de recréer ces classes, alors qu'elles sont à portée de main.

```
>>> from collections import Counter
>>> names = ['Alice', 'Bob', 'Bob', 'Alice', 'Alex', 'Bob']
>>> count = Counter(names)
>>> count
Counter({'Bob': 3, 'Alice': 2, 'Alex': 1})
>>> count['Alice']
2
>>> count['Camille']
0
```

Viennent ensuite les autres modules, tels que `itertools`, `functools` ou `operator`. Ces modules regroupent divers utilitaires sympathiques, qui simplifient grandement le code. En faire bon usage permet de se conformer aux standards du langage.

```
>>> from itertools import product
>>> for x, y in product(range(10), range(5)):
...     print('{} + {} = {}'.format(x, y, x + y))
...
0 + 0 = 0
0 + 1 = 1
...
9 + 3 = 12
9 + 4 = 13
```

```
>>> import functools, operator
>>> add_3 = functools.partial(operator.add, 3)
>>> add_3(5)
8
```

Enfin, suivant le domaine d'application du projet, entrent en compte les modules dédiés : `re`, `math`, `random`, `urllib`, `datetime`, `struct`, etc., et leurs propres bonnes pratiques, souvent détaillées dans la documentation.

La référence complète de la bibliothèque standard peut être trouvée ici.

6. Les bons réflexes

En premier lieu, il faut bien sûr se relire en faisant attention aux divers principes et règles énoncés. Voire se faire relire par un tiers lorsque cela est possible.

Vous l'aurez compris, un autre réflexe sera de s'imprégner de la bibliothèque standard, et de s'assurer pour chaque fonctionnalité que l'on s'apprête à implémenter que celle-ci n'y existe pas déjà.

La fonction `help` permet aussi d'obtenir plus d'informations sur un module, un type, une fonction, ou même un mécanisme du langage. Elle offre ainsi un accès rapide à la documentation directement depuis votre interpréteur interactif. Utilisée sans paramètre, `help` propose aussi une aide interactive.

```
>>> import itertools
>>> help(itertools)
>>> help(str)
>>> help(max)
>>> help('for')
```

```
>>> help()
```

Il conviendra aussi de connaître les bibliothèques tierces, dans une moindre mesure, afin de savoir trouver une bibliothèque répondant à un besoin précis. Il n'est pas nécessaire de toutes les connaître sur le bout des doigts, ni de sortir une usine à gaz pour une petite fonctionnalité, mais simplement de ne pas réinventer la roue.

Les paquets Python sont généralement publiés sur le PyPI , ce qui en fait un répertoire de choix pour trouver une bibliothèque.

Maintenant, oui.

J'espère que par cet article vous aurez appronfondi votre connaissance du langage, et reconnaîtrez simplement un code pythonique d'un autre.

Cet article ne peut être exhaustif, et certains points restent probablement encore flous. Mais les idiomes viennent avec le temps, par la pratique.