Sciences et technologies

Avenue V. Maistriau 8a
B-7000 Mons
+32 (0)65 33 81 54
scitech-mons@heh.be

WWW.HEH.BE

Programmations : Normes

Cours d'aide à la programmation en Python

Bachelier en informatique– Finalité Télécommunications et réseaux – Bloc 1.

Rédigé par :
Desmet Erwin – erwin.desmet@heh.be

Table des matières

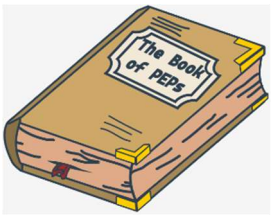
1. Introduction au PEPS

1.1 Qu'est-ce que PEP?

- PEP peut faire référence à:
- les fans de football l'associeront certainement au célèbre ex-footballeur et entraîneur de football, Josep Guardiola, dont le surnom est (devinez quoi) *Pep*;
- ceux qui ont un emploi médical penseront sûrement à la *prophylaxie post-exposition*, qui à son tour a à voir avec la prise de mesures médicales préventives après un contact avec des agents pathogènes;
- Les médecins penseront à la *réaction en chaîne proton-proton*;
- certains joueurs plus âgés (?) visualiseront l'image de *Pep, le chien,* un personnage amusant d'un jeu informatique Atari populaire pour enfants du début des années 90;
- tandis que les programmeurs Python pointeront immédiatement vers un document en ligne, qui décrit les normes du langage et fournit des informations sur de nombreux changements et processus liés à Python.

Dans ce module, comme vous pouvez vous y attendre, nous allons nous concentrer sur le dernier des nombreux PEP.

1.2 Dans la jungle des PEPs



Il y a beaucoup de PEP, des centaines. Consultez https://peps.python.org/
pour découvrir par vous-même que ce ne sont pas des informations inutiles.

Il serait remarquable, mais malheureusement c'est un trop gros défi, de les couvrir tous dans ce cours. Pour cette raison, nous en avons choisi quatre qui méritent une analyse plus approfondie par apport à vos besoins et qui devraient être considérés comme des lectures incontournables. Il s'agit de :

- PEP 1 – Objectif et lignes directrices du PEP, qui fournit des renseignements sur l'objectif des PEP, leurs types et présente des lignes directrices générales;
- PEP 8 – Guide de style pour le code Python, qui donne les conventions et présente les meilleures pratiques pour le codage Python;
- PEP 20 – Le Zen de Python, qui présente une liste de principes pour la conception de Python;
- PEP 257 – Docstring Conventions, qui fournit des lignes directrices pour les conventions et la sémantique associées aux docstrings Python.

Nous vous encourageons à vous lancer dans les PEPs par vous-même. Nous sommes sûrs que vous deviendrez de plus en plus curieux à leur sujet à mesure que votre expérience de programmation et votre sensibilisation grandiront.

1.3 PEP 1 – Objectif et lignes directrices du PEP

PEP est un acronyme qui signifie **Python Enhancement Proposals**, qui, en fait, **est une collection de directives, de meilleures pratiques, de descriptions de (nouvelles) fonctionnalités et implémentations, ainsi que de processus, de mécanismes et d'informations importantes** entourant Python.
En termes simples, si une nouvelle fonctionnalité est prévue pour être ajoutée à Python, elle sera détaillée dans un PEP avec les spécifications techniques et la justification de sa mise en œuvre.
C'est à cela, entre autres, que sert le PEP.

Il existe trois types différents de PEP :

- **Standards Track PEP**, qui décrivent les nouvelles fonctionnalités et implémentations du langage;
- **Les PEP informatifs**, qui décrivent les problèmes de conception de Python, ainsi que des lignes directrices et des informations à la communauté Python ;
- **Les PEP de processus**, qui décrivent divers processus qui tournent autour de Python (par exemple, proposer des changements, fournir des recommandations, spécifier certaines procédures).

1.4 PEP 1 – Objet et lignes directrices du PEP suite

Les PEP s'adressent principalement aux développeurs Python et aux membres de la communauté Python. Ils sont conservés sous forme de fichiers texte dans un référentiel et peuvent être consultés en ligne à https://www.python.org/dev/peps/.

Mais saviez-vous que vous pouvez également proposer votre propre PEP? Si vous avez une nouvelle (brillante) idée pour Python, vous êtes plus que bienvenu pour devenir le **champion** de votre PEP, c'est-à-dire celui qui rédige une proposition PEP, la met en discussion dans des forums liés au sujet et tente de parvenir à un consensus communautaire à ce sujet.
Les formats PEP, les modèles et le processus de soumission (y compris le signalement des bogues et la soumission des mises à jour) ainsi que les étapes subséquentes : examen, résolution et maintenance, sont tous décrits en détail dans https://www.python.org/dev/peps/pep-0001/#start-with-an-idea-for-python .

Enfin et surtout, PEP 1 définit :

- *le conseil directeur de Python*, c'est-à-dire un comité de cinq personnes qui sont les autorités finales et qui acceptent ou rejettent les PEP;
- *Les développeurs principaux de Python*, c'est-à-dire le groupe de bénévoles qui gèrent Python;
- *Le BDFL de* Python, c'est-à-dire Guido van Rossum, le créateur original de Python, qui a servi de Benevolent **D**ictator **F**or **L**ife du projet jusqu'en 2018, date à laquelle il a démissionné du processus décisionnel.

2. PEP 20 – Le zen de Python

Le **Zen** de Python est une collection de 19 aphorismes, qui reflètent la philosophie derrière Python, ses principes directeurs et son design.

Tim Peters, est un contributeur majeur et de longue date du langage de programmation Python et à la communauté Python, a écrit ce poème de 19 lignes sur la liste de diffusion Python en 1999, et il est devenu l'entrée #20 dans les propositions d'amélioration Python en 2004.
C'est l'un des *œufs de Pâques* (c'est-à-dire des messages ou des fonctionnalités cachés, secrets) inclus dans l'interpréteur Python.

Voyons maintenant la magie. Allez dans la fenêtre de votre éditeur, tapez **import this,** exécutez le code, et voilà! Pouvez-vous voir ce qui se passe?

Ce que vous voyez est une collection de quelques vérités générales pour les règles de conception et la prise de décision Python. Même si le « poème » semble être imprégné de contradictions et d'allusions, nous vous assurons que les aphorismes sont extrêmement pratiques et de bon sens, et vous êtes encouragés à les accepter et à les mettre en œuvre dans votre code.

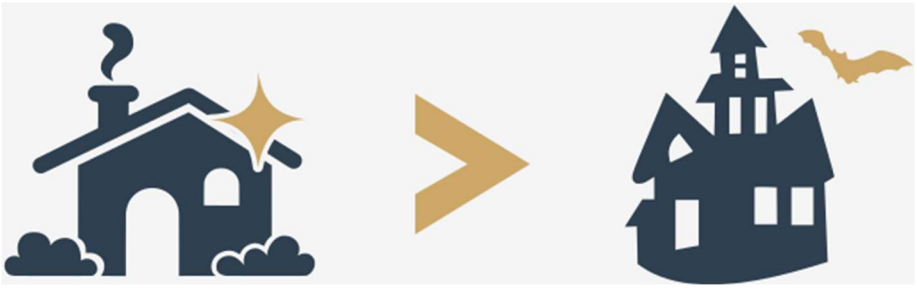
Ceux-ci, bien sûr, devraient être considérés de manière holistique, plutôt qu'individuellement, mais, quand même, essayons de méditer sur chacun d'eux.

2.1 Beau vaut mieux que laid

La beauté est une expérience plutôt subjective. Mais, comme l'a dit Emmanuel Kant, **l'expérience esthétique même de la beauté est un jugement de la vérité humaine**.

Et même si l'ordinateur ne se soucie pas de la beauté ou de l'esthétique, les gens le font, et nous devons nous rappeler qu'un programme bien écrit est non seulement plus agréable à lire, mais aussi plus **lisible**.

Python a certaines **règles de style** qu'il est recommandé aux programmeurs de suivre. Il s'agit, entre autres, d'une longueur de ligne maximale de 79 caractères, de conventions de dénomination de variables, de placer des instructions sur des lignes séparées et bien d'autres.



Exemple : Écrivez un programme qui calcule l'hypoténuse d'un triangle rectangle.

```
from math import sqrt
sidea = float(input("The length of the 'a' side:"))
sideb = float(input("The length of the 'b' side:"))
sidec = sqrt(a**2+b**2)
print("The length of the hypotenuse is", sidec )
```

```
from math import sqrt

side_a = float(input("The length of the 'a' side: "))
side_b = float(input("The length of the 'b' side: "))
hypotenuse = sqrt(a**2 + b**2)

print("The length of the hypotenuse is", hypotenuse)
```

2.2 Explicite est mieux qu'implicite

Le code que vous écrivez doit être **explicite et lisible**.
Chaque fois que vous souhaitez utiliser une caractéristique implicite de la langue, demandez-vous si vous en avez vraiment besoin. Peut-être y a-t-il une meilleure façon d'implémenter la fonctionnalité. Sinon, pensez à laisser un commentaire dans le code pour expliquer ce qui se passe afin que les autres programmeurs aient plus facile à comprendre votre code.

En Python, il est préférable d'utiliser non seulement la manière la plus simple d'exprimer une idée de programmation, mais aussi la plus explicite, concrète, spécifique.

Par conséquent, c'est parfois une bonne idée d'ajouter de la verbosité à votre code car tout compte pour la lisibilité. Donner des noms de variables et de fonctions explicites, ou ajouter soyez explicite pour vos importations ou aux arguments de fonction sera une bonne pratique.

Exemple : Importer des **pommes** et des **bananes** à partir du module **fruit.py**.

```
from fruit import *

apples(2, 3.45)
```

```
from fruit import apples, bananas

apples(quantity=2, price=3.45)
```

2.3 Simple vaut mieux que complexe

La simplicité est la clé du succès.

Une solution **plus simple** est généralement préférée à une solution complexe, et généralement, l'approche minimaliste l'emporte. N'oubliez pas : utilisez **des outils adaptés à la spécificité** de votre projet.

Utiliser un avion pour vous transporter dans un magasin voisin pourrait être acceptable (en supposant que vous soyez légèrement excentrique), mais il suffit généralement de marcher ou de conduire. De même, vous ne marcheriez normalement pas la distance si vous vouliez voyager du Royaume-Uni aux États-Unis. Prendre l'avion serait une idée plus sensée ici.

Envisagez de ne pas adopter une approche orientée objet lorsque ce n'est pas nécessaire. Utilisez moins de lignes de code si cela est possible.

Si vous devez mettre en œuvre une solution plus complexe, **divisez les problèmes en parties plus petites et plus simples**.

Exemple : Triez la liste des **numéros** par ordre croissant.

```
import heapq

numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
heapq.heapify(numbers)

sorted_numbers = []

while numbers:
    sorted_numbers.append(heapq.heappop(numbers))

print(sorted_numbers)
```

```
numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
numbers.sort()

print(numbers)
```

Remarques : Lors de l'examen, il arrive souvent qu'on interdise les fonctions types et donc le complexe l'emporte cette fois. Etudes ne rime pas toujours avec concrétisation professionnel.

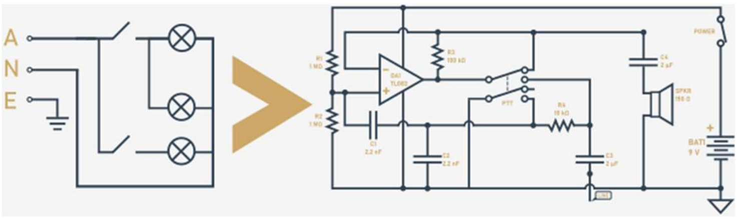
2.4 Complexe vaut mieux que compliqué

Lorsque des solutions simples ne sont pas possibles, soyez conscient des **limites de la simplicité** et utilisez plutôt des solutions complexes.

Distinguer la forme complexe comme un composé de nombreux éléments et compliqué comme difficile à comprendre, est une autre chose à considérer lors de l'écriture de code qui est bien plus difficile à mettre en place.

En d'autres termes, il y a des moments où une solution complexe peut être préférée à une solution simple, en particulier dans le cas où cette dernière cause un malentendu, un doute ou une mauvaise interprétation. Vous devriez alors éviter la méthode de simplification.

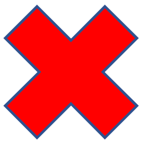
D'autre part, complexe est toujours préféré à compliqué. Lorsque votre code devient volumineux et trop difficile à comprendre et à saisir, **divisez-le en parties bien séparées**, afin qu'il soit plus facile à gérer et à manipuler.



Évitez les mauvaises compréhensions, le manque de clarté et les malentendus.

Exemple : effectuez cinq additions de deux nombres.

```
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
```

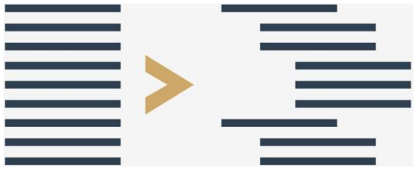


```
def addition(x, y):
    print(x, "+", y, "=", x+y)

for i in range(5):
    first_number = int(input("Enter the first number: "))
    second_number = int(input("Enter the second number: "))
    addition(first_number, second_number)
```

2.5 Déroulé est mieux qu'imbriqué

Le code imbriqué le rend plus difficile à suivre et à comprendre. Imbriquer deux ou trois niveaux de profondeur peut encore être bon, mais tout ce qui va au-delà devient déroutant et illisible.

Même si vous pouvez réellement avoir n'importe quel niveau de boucles imbriquées ou d'instructions en Python, **tout ce qui dépasse trois** niveaux devrait être un signal clair que c'est peut-être le bon moment pour commencer à refactoriser votre code.



Le code « plat » est plus convivial et devient beaucoup **plus facile à maintenir**.

Exemple : affichez un message, selon que x se situe ou non dans la plage comprise entre 4 et 6.

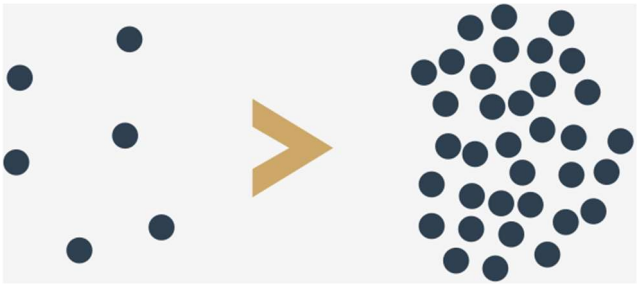
```
x = float(input("Enter a number: "))

if x > 0:
    if x > 1:
        if x > 2:
            if x > 3:
                if x >= 4:
                    if x <= 6:
                        print("x is a number between 4 and
6.")
else:
    print("x is not a number between 4 and 6.")
```

```
x = float(input("Enter a number: "))

if x >= 4 and x <=6:
    print("x is a number between 4 and 6.")
else:
    print("x is not a number between 4 and 6.")
```

2.6 Clairsemé vaut mieux que dense

N'écrivez pas trop de code sur une ligne, n'insérez pas trop d'informations dans une petite quantité de code, n'écrivez pas de lignes de code trop longues, utilisez les espaces blancs de manière responsable : tout cela affecte la lisibilité et la compréhension de votre programme.

Le moyen le plus simple et le plus courant d'introduire de la parcimonie dans votre code consiste à introduire l'imbrication. C'est probablement pourquoi cet aphorisme vient juste après celui qui nous dit de préférer le code plat au code imbriqué. La clé de la lisibilité est de trouver un équilibre entre les deux : réduire **l'imbrication**, puis essayer de **réduire la densité**.



Exemple : Imprimez le message « Hello, World! » si la valeur passée à la variable x est égale à 1.

```
x = 1
if x == 1 : print("Hello, World!")
```

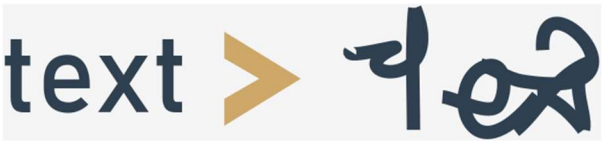
```
x = 1
if x == 1:
    print("Hello, World!")
```

2.7 La lisibilité compte

Votre code n'est pas seulement lu par les ordinateurs, il est aussi (ou surtout) lu par les humains. En fait, c'est l'essence de la philosophie Python, et l'ensemble de la conception et de la culture Python tourne autour de l'affirmation même que **« le code est lu plus souvent qu'il n'est écrit »** (Guido Van Rossum).

Tous les aphorismes précédents (et les suivants) ouvrent la voie à la lisibilité, dans une plus ou moins grande mesure, comme l'un des facteurs les plus cruciaux à garder à l'esprit lors de la création du code. Chaque fois que vous êtes tenté d'abandonner la lisibilité, que ce soit pour gagner du temps en ayant à imaginer des noms significatifs, l'effort fait pour formater votre code ou toute autre raison : rejetez la tentation.
Ne sous-estimez pas le pouvoir de la lisibilité, en particulier lorsque vous devez revenir à votre code après un certain temps, ou laisser le code à d'autres pour le développer à l'avenir.

Donner des **noms significatifs** aux variables, fonctions, modules et classes; **styliser correctement les blocs de code**; **Utiliser des commentaires** si nécessaire; garder votre code propre et élégant : tout cela contribue à la lisibilité et à la convivialité de votre code.



Rappelez-vous: la lisibilité de votre code reflète à quel point vous êtes un programmeur responsable. Non seulement cela reflète bien la qualité du code, mais cela reflète bien son auteur.

Exemple : Écrivez un programme qui calcule le prix brut d'un produit.

```
def f(i):
    l = i + (0.08 * i)
    return l
```

```
# Calculates the gross price of products in Wonderland.

def calculate_gross_price(net_price):
    gross_price = net_price + (0.08 * net_price)
    return gross_price
```

2.8 Les cas particuliers ne sont pas assez spéciaux pour enfreindre les règles...

La discipline, l'uniformité et la conformité aux normes et aux conventions sont tous des éléments importants de l'élaboration professionnelle et responsable des codes. Il ne devrait y avoir aucune exception qui nous permette d'enfreindre les principes régissant les meilleures pratiques de codage. Aucun cas particulier tel que la pression du temps ou la complexité d'un problème donné ne devrait être une excuse pour écrire du code qui ne suit pas les directives.

Ce n'est pas seulement une question de lisibilité, bien que cela devrait être l'une des premières choses auxquelles vous pensez, mais il s'agit aussi de s'en tenir aux décisions liées à la conception et au développement que vous avez prises, qu'il s'agisse de cohérence pour **assurer la rétrocompatibilité**, de garder les conventions de nommage inchangées, ou quoi que ce soit d'autre.

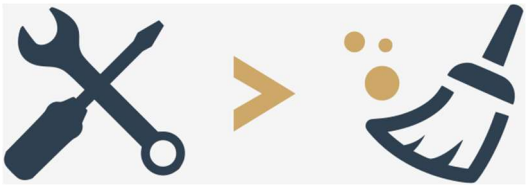
Exemple : écrivez une fonction qui multiplie deux nombres et une fonction qui ajoute deux nombres.

```
def multiply_two_numbers(first_number, second_number):
    return first_number * second_number

print(multiply_two_numbers(7, 9))


def addingTwoNumbers(firstNumber, secondNumber):
    return firstNumber + secondNumber

print(addingTwoNumbers(7, 9))
```

```
def multiply_two_numbers(first_number, second_number):
    return first_number * second_number

print(multiply_two_numbers(7, 9))


def add_two_numbers(first_number, second_number):
    return first_number + second_number

print(add_two_numbers(7, 9))
```

2.9 … Bien que l'aspect pratique l'emporte sur la pureté

D'accord... Qu'est-ce qui se passe? L'aphorisme précédent nous encourageait à ne jamais enfreindre les règles, alors que celui-ci dit qu'il pourrait y avoir des exceptions à cela. Pourquoi ?



Eh bien, nous devons nous rappeler que le but ultime est de résoudre des problèmes réels et d'écrire du code qui effectue une tâche particulière (attendue). Si votre code est élégant, lisible et conforme à toutes les conventions de style importantes, mais ne fonctionne pas comme il le devrait, alors cela n'a pas beaucoup de sens, n'est-ce pas ?

Si les avantages possibles (p. ex., une meilleure performance) sont plus importants que les effets négatifs possibles (p. ex., maintenabilité affectée), les problèmes de codage réels peuvent trouver une excuse pour faire une exception aux règles. L'aspect pratique devient alors plus important que la pureté.

Si vous devez écrire une longue ligne de code de 85 caractères parce que la diviser en deux lignes distinctes affecte la lisibilité, faites-le. Si vous devez conserver la compatibilité avec le code précédemment écrit et utiliser le CamelCase au lieu du snake_case, faites-le. Les règles doivent parfois être enfreintes, des exceptions doivent être faites.

2.10 Les erreurs ne doivent jamais être passé sous silence…

« … À moins qu'elles ne soient explicitement réduites au silence. »

Analysez une situation potentiellement dangereuse ci-dessous :

```
number = input("Enter a number: ")
multiply_number_by_two = number * 2

print("Your number multiplied by two is:",
multiply_number_by_two)
```

Supposons que le programmeur ait oublié de convertir la valeur attribuée à la variable **number** en ***int*** ou ***float***. Le programme ne se bloquera pas. Au contraire, il fonctionnera sans aucun problème et produira un bon résultat. Bien que, loin de celui attendu.
Si l'extrait ne constitue qu'une infime partie de l'ensemble du code, le programmeur peut avoir du mal à trouver la source d'une erreur et à déboguer le programme car aucun message d'erreur explicite n'est affiché dans la console.

Apportons quelques modifications et améliorons un peu l'extrait :

```
number = int(input("Enter a number: "))
multiply_number_by_two = number * 2

print("Your number multiplied by two is:", multiply_number_by_two)
```

Que se passe-t-il lorsque l'utilisateur entre 3,5 ou deux comme entrée ? Eh bien, Python vous informera bien sûr haut et fort qu'il y a eu quelque chose qui ne va pas: il va déclencher l'exception ValueError.

Le langage Python fournit un très bon mécanisme pour la gestion des erreurs, avec un certain nombre d'exceptions intégrées, et un excellent ensemble d'outils pour créer des systèmes de gestion des exceptions définis par l'utilisateur.

Le Zen de Python nous rappelle gentiment que si un bloc de code est incapable de remplir sa fonction et de fonctionner de la manière attendue par le programmeur, il doit arrêter le programme et / ou annoncer haut et fort que quelque chose a mal tourné (c'est-à-dire soulever une exception) plutôt que de continuer à fonctionner sans interruption.

Un programme qui se bloque est **plus facile à déboguer qu'un** programme qui fait taire une erreur. Le fait de soulever une exception **attire votre attention sur le problème et fournit des informations importantes sur** ce qui s'est passé et pourquoi. Les erreurs qui passent silencieusement peuvent infecter le programme et modifier son fonctionnement de sorte qu'il devient imprévisible, inattendu et indésirable.

L'une des tâches les plus difficiles qu'un programmeur ait à faire est de penser à tous les contextes possibles (ou au moins autant d'entre eux que possible) dans lesquels une exception peut se produire. Servir ces exceptions et fournir un remède aux erreurs attendues (et bien gérées) est un défi important, mais en même temps une responsabilité cruciale d'un bon programmeur professionnel. Ne jamais supposé qu'un utilisateur fera ce qu'il doit faire !

Exemple : erreur explicitement silencieuse (à l'aide du mot-clé except). Cependant, l'exception est traitée de manière trop large :

```
try:
    print(1/0)
except Exception as e:
    pass
```

Une version améliorée, gérant un type spécifique d'erreur serait :

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Don't divide by zero!")
```

Eh bien, naturellement, il peut y avoir des situations où vous ne voulez pas crier « Hé! Il y a une erreur! » mais plutôt la gérer de manière subtile et ne pas nécessairement faire des histoires à ce sujet pour l'utilisateur.

Analysez le code ci-dessous dans lequel nous gérons une exception en ajoutant une valeur par défaut :

```
try:
    number = int(input("Enter an integer number: "))
except:
    number = 0
```

2.11 Face à l'ambiguïté, refusez la tentation de deviner

Les suppositions fonctionneront sûrement dans de nombreux cas, mais dans beaucoup d'autres, elles peuvent vous décevoir amèrement. Cette directive transmet un double message : d'une part, elle vous dit d'avoir une confiance limitée dans le code que vous écrivez, tandis que d'autre part, elle implique que vous devriez avoir une confiance limitée dans le code que vous lisez. Mais qu'est-ce que cela signifie?

La première chose à retenir est de toujours **tester votre code** avant de le publier en production et de le déployer chez les clients. Cela semble évident et raisonnable? Eh bien, oui, mais souvent, les programmeurs négligent ou oublient cette habitude simple, que ce soit parce qu'ils font confiance à leurs compétences en codage dans la mesure, par exemple, où ils rejettent toute possibilité de faire des fautes de frappe, ou parce qu'ils travaillent sous une grande pression de temps et sentent qu'ils n'ont pas le temps de tester.

Une chose importante à garder à l'esprit est que **tester votre code vous permet de gagner du temps**, pas de le gaspiller. Si vous trouvez un bogue à un stade précoce, il vous en coûtera moins de temps et d'argent pour le corriger. Si vous ne testez pas votre code et qu'il s'avère qu'il y a un bogue à un stade avancé de développement, les corrections peuvent être une entreprise assez coûteuse et chronophage. Vous verrez dans d'autres parties de cours que les tests c'est la base de la pyramide d'un bon développeur.

Une autre chose est que vous devriez **éviter est d'écrire du code ambigu**, ce qui signifie que vous ne devriez laisser aucune place pour la supposition. Donnez à vos variables des **noms auto-commentés** et **laissez des commentaires** si nécessaire (et seulement si). Si vous importez un module, rendez-le explicite. Si un extrait particulier est complexe ou compliqué, expliquez son fonctionnement. Ne laissez jamais de commentaires ou n'utilisez jamais de noms faux, confus ou trompeurs!

De même, si vous soupçonnez qu'il y ait quelque chose qui ne va pas dans le code que vous lisez, ou si vous sentez qu'il y a quelque chose de peu clair dedans, ne devinez pas son fonctionnement : testez-le!

Analysons l'exemple suivant :

```
fun(1, 2, 3)
fun(a=1, b=2, c=3)
```

Les deux invocations de fonction peuvent sembler identiques, mais ce n'est pas nécessairement. Il n'est pas possible de le savoir sans voir la définition de la fonction.
Si la définition de la fonction est comme celle ci-dessous, les résultats peuvent différer :

```
def fun(x=0, y=0, z=0, a=1, b=2, c=3):
    pass
```

Vous voyez pourquoi ?

Regardons un deuxième exemple :

```
print("A" > "a")
print(1.0 == 1)
print("1" == 1)
print(True == "1")
print(True == 1)
print(True == 1.0)
print("1" + "1")
print(1 + 1)
print(1 + "1")
```

Connaissez-vous le résultat de l'extrait ci-dessus? Êtes-vous certain de cela, ou vous tentez de deviner ? Les comparaisons et expressions ci-dessus fourniraient-elles les mêmes résultats dans différents langages de programmation ? Eh bien, pas nécessairement...

Si vous travaillez sur un programme qui accepte des données de l'utilisateur, ne vous fiez surtout pas à vos suppositions, car **ce que vous supposez être le plus courant peut s'avérer le moins courant face aux** données réelles des utilisateurs qui sont parfois un peu farfelu.

Par exemple, si vous écrivez une application qui accepte du texte de l'utilisateur, spécifiez le codage que vous attendez de lui et acceptez uniquement cet encodage particulier, en gérant tous les cas que l'encodage attendu ne fournit pas. Si vous devez effectuer une conversion, utilisez des outils spécialisés et valides pour éviter les brouillages de caractères ou les plantages de programme.

N'oubliez pas de toujours rechercher les contextes dans lesquels votre programme pourrait planter et de les diffuser. Ne vous fiez pas à vos suppositions ou à votre conviction que l'utilisateur suivra strictement vos instructions. Analysez le fragment d'une simple session Python interactive que nous avons fournie ci-dessous. Pouvez-vous voir ce qui a mal tourné?

```
>>> integer_number = int(input("Enter an integer number: "))
Enter an integer number: 15.6
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    integer_number = int(input("Enter an integer number: "))
ValueError: invalid literal for int() with base 10: '15.6'
```

2.12 Il devrait y avoir une (et de préférence une seule) façon évidente de le faire

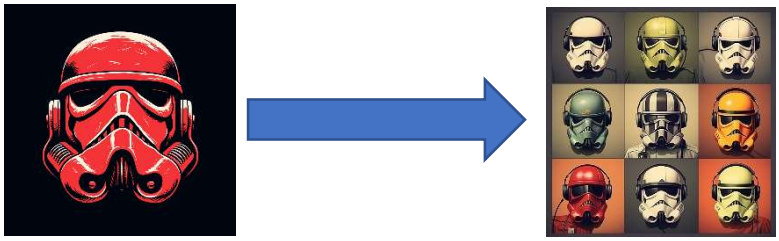
« Bien que cette façon de faire ne soit pas évidente au début, sauf si vous êtes Néerlandophone. »

Il peut y avoir plusieurs façons d'atteindre le même objectif. Par exemple, si vous souhaitez prendre le prénom et le nom de l'utilisateur et les afficher à l'écran, vous pouvez le faire de l'une des manières suivantes :

```
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")

print("Your name is:", first_name, last_name)
print("Your name is:" + " " + first_name + " " + last_name)
print("Your name is: {} {}".format(first_name, last_name))
```

Lequel d'entre eux est votre préféré? Cela dépend de ce que vous voulez réaliser, comment vous voulez formater le texte en sortie, quelles conventions passées ont été utilisées, etc.

Il semble qu'il n'y ait rien de mal à avoir plusieurs façons de faire une certaine chose, tant que nous sommes prêts pour la confrontation et que nous sommes capables de **nous mettre d'accord sur la meilleure façon d'atteindre un objectif particulier**.



Quel est le meilleur casque le rouge ou … un autre ? Soyons pareil !

La ligne directrice nous rappelle également que c'est une bonne idée de **suivre les normes et les conventions d'utilisation de la langue**. Par exemple, si vous avez utilisé snake_case pour nommer vos variables dans votre code jusqu'à présent, il peut être une mauvaise idée de commencer à utiliser CamelCase pour le reste de votre code. Eh bien, à moins que vous ne le fassiez dans un but spécifique, et les avantages d'une telle approche sont plus importants que les inconvénients.

Enfin, l'aphorisme fonctionne comme une indication douce d'un autre conseil important: dans la mesure du possible, il est bon de se rappeler que **chaque fonction, chaque classe, chaque méthode – chaque entité – devrait avoir une seule responsabilité cohésive**. Pourquoi? Parce qu'une telle approche vous aide à gagner en clarté et à produire un code plus propre, le rend plus facile et moins coûteux à maintenir, et moins vulnérable aux bogues.

En ce qui concerne la deuxième partie de l'aphorisme, d'une part, il s'agit d'une blague : les Néerlandais ont une façon différente de penser, une vision du monde différente et une façon différente de faire les choses (vous vous souvenez certainement que Guido van Rossum est également néerlandais).

D'un autre côté, cependant, cela indique que trouver la meilleure solution peut être un processus long et difficile: une façon évidente de faire quelque chose peut ne pas nécessairement être évidente au début. Trouver une solution pertinente et à privilégier peut demander du temps, des efforts et de changer certaines habitudes. Ça tombe bien, vous n'êtes pas encore sensé en avoir !

Python lui-même en est un bon exemple : il évolue toujours, ses fonctionnalités et les idées qui l'entourent changent, et les programmeurs Python peuvent encore percevoir des choses relativement similaires différemment les uns des autres.
Quelle est la meilleure façon d'accéder aux valeurs d'un dictionnaire : en utilisant la méthode get(), ou la syntaxe my_dict['key'], ou d'une autre manière encore?
Quelle est la meilleure façon de lire un fichier : bloc par bloc, ligne par ligne ?
Quelle est la meilleure façon d'imprimer le prénom et le nom de l'utilisateur à l'écran...?

2.13 Maintenant c'est mieux que jamais

« Bien que jamais est souvent mieux que tout de suite. »

Vous ne devriez pas remettre à demain ce que vous pouvez faire aujourd'hui. C'est un proverbe bien connu. Pourquoi? Eh bien, parce qu'il n'y a jamais de bon moment pour quoi que ce soit – il y a toujours des « mais » et des « si » qui vous disent d'attendre plus longtemps et donc de retarder les choses. Avant de vous mettre à faire ces choses – écrire votre code – vous aurez peut-être oublié les idées ou les informations dont vous aviez besoin pour bien le faire.

Et attention spoiler alerte !!! C'est pareil avec vos études !!!

Python vous permet de traduire rapidement vos idées en code fonctionnel. Chaque fois que vous ressentez l'effet **eureka** ou que vous avez votre moment d'inspiration, écrivez vos pensées et encodez-les en Python (ou au moins utilisez une forme de pseudo-code), même si votre code est loin d'être parfait. Vous pouvez ensuite l'affiner, le développer ou le refactoriser très facilement.

Une autre chose à retenir est qu'il n'existe pas de chose parfaite. Vous pouvez travailler dur pour vous rapprocher de la perfection, affiner votre code, le refactoriser plusieurs fois, mais il ne sera jamais parfait. Même vos éminent professeurs ne sont pas parfait ! Je sais vous tomber de haut !!

Rien ne le peut, et vous devez en être conscient. Si vous cédez à la tentation de terminer un programme et de le libérer seulement quand il est parfait, il est fort probable que vous ne le ferez jamais.

Votre programme a le fonctionnement attendu? Il passe tous les tests? Peut-être est-il prêt pour que le monde le voie?

D'autre part, l'aphorisme nous dit de ne pas oublier le bon équilibre. Tout comme la perfection est l'ennemi du bien, il s'avère souvent que plus rapide est l'ennemi du plus lent. Il y a des cas où les choses ne devraient pas être précipitées.

Votre fonction ne fonctionne pas comme prévu et vous ne pouvez pas la réparer aujourd'hui? Marquez-le comme obsolète afin de ne pas l'oublier :

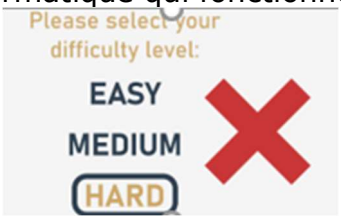
```
def deprecated_function():
    raise DeprecationWarning
```

Votre projet doit passer par la phase de test? Avez-vous besoin de recueillir les commentaires des utilisateurs ? La campagne marketing n'est pas encore prête ? Prenez le temps de tout faire correctement et de lancer le produit quand il est vraiment prêt, pas quand il semble prêt.

2.14 Si la mise en œuvre est difficile à expliquer, c'est une mauvaise idée

« Si la mise en œuvre est facile à expliquer, cela peut être une bonne idée. »

Tout ce qui peut être **expliqué avec des mots** peut être **traduit en code**, et finalement transformé en un programme informatique qui fonctionne bien.



Si vous pouvez expliquer ce que vous attendez d'un programme, ce que vous voulez qu'il fasse : un tel programme peut être conçu.
Si vous avez du mal à expliquer ses caractéristiques et ses fonctionnalités, cela peut indiquer que votre idée devrait peut-être être repensée et digérée.

La simplicité et le minimalisme sont à nouveau les clés (s'ils ne tuent pas la lisibilité).
Simple vaut mieux que complexe, mais complexe vaut mieux que compliqué : si vous avez déjà oublié, voici un rappel subtil.
Gardez les choses simples; Plus c'est simple, mieux c'est.
Cependant, même si quelque chose est facile à expliquer, cela ne signifie pas que c'est bon. Il est simplement **plus facile de juger** si c'est le cas ou non. En cas de doute, faites examiner votre mise en œuvre par vos pairs et voyez combien d'efforts il leur faut pour saisir l'idée et comprendre l'ensemble du concept. Une autre paire d'yeux peut jeter un nouvel regard sur votre projet et vous aider à le voir donc d'une nouvelle manière.

2.15 Les espaces de noms sont une excellente idée – faisons-en plus !

Python fournit un bon mécanisme d'espace de noms bien organisé pour gérer la disponibilité des identificateurs que vous souhaitez utiliser et **éviter les conflits avec des noms déjà existants** dans différentes étendues.

Qu'est-ce qu'un espace de noms ? D'une manière générale, il s'agit d'un « mappage des noms d'objets »(https://docs.python.org/3/tutorial/classes.html) implémenté en Python sous la forme d'un dictionnaire.

Qu'est-ce que cela signifie? En termes simples, cela signifie que chaque fois que vous définissez une variable, Python « se souvient » de deux choses: l'identifiant de la variable et la valeur que vous lui transmettez.

Comment cela se passe-t-il? Python les ajoute implicitement à un dictionnaire interne qui possède une portée particulière, c'est-à-dire la région d'un programme Python où les espaces de noms sont accessibles. Si vous souhaitez accéder à cette variable, Python recherche son nom dans le dictionnaire et renvoie la valeur qui lui a été transmise. Si la variable n'existe pas et, par conséquent, n'est pas trouvée, elle déclenche l'exception NameError.

Fonctions, classes, objets, modules, paquets... Ce sont tous des espaces de noms. Ce fait se traduit par ce qui suit : un espace de noms plus spécifique ne peut pas être modifié par un espace de noms moins spécifique, car ils résident dans deux étendues différentes (par exemple, une variable locale à l'intérieur d'une fonction n'influence pas une variable globale[1]). Toutefois, un espace de noms plus spécifique a accès à un espace de noms moins spécifique (par exemple, une variable globale est accessible à partir d'une fonction).

Utilisez les espaces de noms pour rendre votre code plus clair et plus lisible. Par exemple, procédez comme suit :

```
from starwars import jedi

jedi.saber(green)
jedi.power(no)
```

```
from starwars.jedi import saber,power

saber(green)
power(no)
```

Pourquoi ? Parce que le premier exemple montrera clairement que saber et power proviennent d'un module différent, et non d'un cadre local.

[1] L'utilisation du mot-clé global avant une variable globale à l'intérieur de la fonction est un mécanisme qui vous permet de modifier cette variable, même si elle réside dans une portée différente (mauvaise pratique).

3 PEP 8 – Les meilleurs pratiques et conventions

3.1 Introduction au PEP 8

Comme mentionné précédemment, PEP 8 est un document qui fournit des **conventions de codage** (code style guide) pour Python.
PEP 8 est considéré comme l'un des PEP les plus importants et une lecture incontournable pour tout programmeur Python professionnel, car il contribue à rendre le code plus cohérent, plus lisible et plus efficace.

Même si certains projets de programmation peuvent adopter leurs propres directives de style (auquel cas ces directives spécifiques au projet peuvent être préférées aux conventions prévues par PEP 8, en particulier en cas de conflits ou de problèmes de rétrocompatibilité), les meilleures pratiques PEP 8 sont toujours fortement recommandées, car elles vous aident à mieux comprendre la philosophie derrière Python et à devenir un programmeur plus conscient et compétent. Ce qui il me semble est le but d'un étudiant.

Le PEP 8 continue **d'évoluer**, d'autres conventions y sont identifiées et incluses, et en même temps, certaines anciennes conventions sont identifiées comme obsolètes et on demande de ne plus les suivre au possible. Et oui come on vous le disait le Python est un langage vivant et évolutif.

3.2 Le lutin des petits esprits

« Une cohérence stupide est le lutin des petits esprits. »
Ceci est une citation de l'essai de Ralph Waldo Emerson « Self-Reliance » où Emerson exhorte les lecteurs à être cohérents dans leurs croyances et leurs pratiques. Dans notre cas, cela signifie que nous ne devons pas oublier une observation simple mais importante: **notre code sera lu beaucoup plus souvent qu'il ne sera écrit**.

D'une part, la cohérence est un facteur crucial qui détermine la lisibilité du code. D'autre part, l'incohérence avec PEP 8 peut parfois être une meilleure option. Si les guides de style ne sont pas applicables à votre projet, il peut être préférable de les ignorer et de décider vous-même ce qui est le mieux.

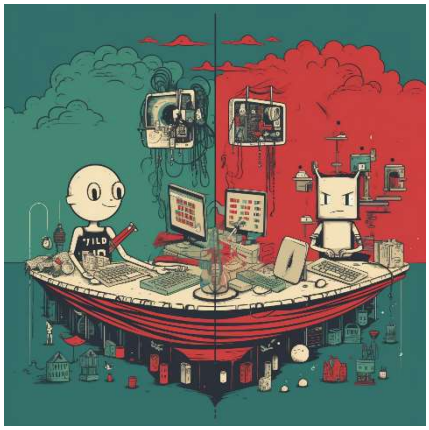
Comme le dit le PEP 8 :

Un guide de style est une question de cohérence. La cohérence avec ce guide de style est importante. La cohérence au sein d'un projet est plus importante. La cohérence au sein d'un module ou d'une fonction est la plus importante. [...] Cependant, sachez quand être incohérent [...]. En cas de doute, utilisez votre meilleur jugement.

Quand devriez-vous **ignorer** certaines directives spécifiques du PEP 8 (ou du moins envisager de le faire)?

- Si les suivre signifie que vous rompez la rétrocompatibilité.
- Si les suivre aura un effet négatif sur la lisibilité du code.
- Si les suivre entraînera une incohérence avec le reste du code. (Cependant, cela peut être une bonne occasion de réécrire le code et de le rendre conforme à PEP 8.)
- S'il n'y a aucune bonne raison de rendre le code PEP 8 conforme, ou si le code est antérieur à PEP 8.

PEP 8 est destiné à améliorer la lisibilité du code et à le « rendre cohérent sur le large spectre du code Python ». Garder votre code Python conforme à PEP 8 est donc une bonne idée, mais vous ne devriez jamais adhérer aveuglément à ces recommandations. Vous devriez toujours utiliser votre meilleur jugement. C'est la que se trouve la différence entre un bon étudiant et un étudiant lambda, entre un bon programmeur et un mec qui code.



3.3 Vérification de la conformité à PEP 8

Il existe de nombreux outils utiles qui peuvent vous aider à valider votre style de code et à le comparer aux conventions de style PEP 8. Ces outils peuvent être installés et exécutés localement, ou accessibles en ligne. Nous allons vous en montrer seulement deux, mais nous vous encourageons à explorer davantage :

- **pycodestyle** (anciennement appelé *pep8*, mais le nom a été changé pour éviter toute confusion) : Est un vérificateur de de style Python; il vous permet de vérifier la conformité de votre code Python avec les conventions de style de PEP 8. Vous pouvez installer l'outil avec la commande suivante dans le terminal:

```
$ pip install pycodestyle
```

Vous pouvez l'exécuter sur un ou plusieurs fichiers pour obtenir des informations sur les non-conformités (et indiquer les erreurs dans le code source et leur fréquence). Pour plus d'informations :
https://github.com/PyCQA/pycodestyle
Documentation : https://pycodestyle.pycqa.org/en/latest/

- Vous pouvez également installer **autopep8** pour formater automatiquement votre code Python afin de le rendre conforme aux directives PEP 8. Pour pouvoir l'utiliser, vous avez besoin de l'installation *pycodestyle* sur votre machine afin d'indiquer les parties du code qui nécessitent des corrections de formatage.

  Pour plus d'informations : https://pypi.org/project/autopep8/

- **PEP 8 online est un vérificateur PEP 8** en ligne créé par Valentin Bryukhanov qui vous permet de coller votre code ou de télécharger un fichier, et de le valider par rapport aux directives de style PEP 8. L'outil en ligne est construit à l'aide de Flask, Twitter Bootstrap et du module PEP8 (le même module que nous venons de décrire).

  Pour plus d'informations : http://pep8online.com/about

3.4 PEP 8 – Schéma des codes

3.4.1 Recommandations pour la mise en page du code

PEP 8 est censé améliorer votre expérience de codage et vous faciliter la vie. Comme indiqué précédemment, la façon dont vous écrivez votre code a un impact important sur sa lisibilité. Cependant, vous ne devez pas oublier qu'il peut également déterminer sa légalité syntaxique.

Dans cette section, nous nous concentrerons sur les recommandations de style liées à des éléments tels que :

- indentation, à l'aide de tabulations et d'espaces;
- longueur de ligne, sauts de ligne et lignes vides;
- Encodage du fichier source et importation de modules.

Indentation

Le niveau d'indentation, compris comme l'espace de début (c'est-à-dire les espaces et les tabulations) au début de chaque ligne logique, est utilisé pour regrouper les instructions.

Lorsque vous écrivez du code en Python, n'oubliez pas de suivre ces deux règles simples:

- Utilisez **quatre espaces par niveau d'indentation**, et;
- Utilisez **des espaces plutôt que des tabulations**.

Cependant, vous pouvez utiliser des tabulations lorsque vous souhaitez **conserver la cohérence** avec le code qui a déjà été mis en retrait avec des tabulations (s'il n'est pas possible ou efficace de le rendre conforme à PEP 8).

Remarque : **Le mélange de tabulation et d'espaces pour l'indentation n'est pas autorisé** dans Python 3. Cela déclenchera une exception TabError: « TabError: utilisation incohérente des tabulations et des espaces dans l'indentation ».

Exemples :

```
# Bad:

def my_fun_one(x, y):
    return x * y

def my_fun_two(a, b):
  return a + b
```

```
# Good:

def my_function(x, y):
    return x * y
```

Lignes de continuation

Les lignes de continuation (c'est-à-dire les lignes de code logiques que vous souhaitez fractionner parce qu'elles sont trop longues ou parce que vous voulez améliorer la lisibilité) sont autorisées si vous utilisez des parenthèses/crochets/accolades :

```
# Bad:

my_list_one = [1, 2, 3,
    4, 5, 6
]

a = my_function_name(a, b, c,
    d, e, f)
```

```
# Good:

my_list_one = [
    1, 2, 3,
    4, 5, 6,
    ]

a = my_function_name(a, b, c,
                     d, e, f)
```

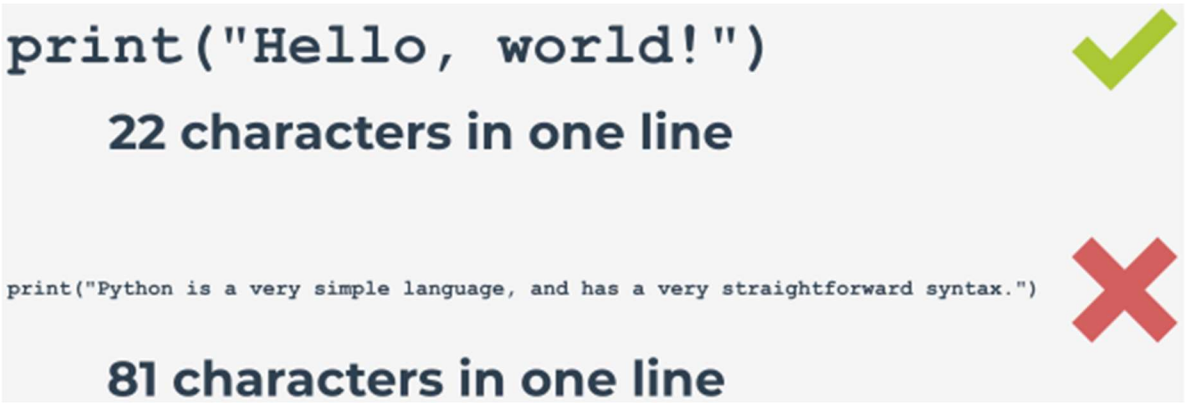
```
# Good:

my_list_two = [
    1, 2, 3,
    4, 5, 6,
]


def my_fun(
        a, b, c,
        d, e, f):
    return (a + b + c) * (d + e + f)
```

Vous pouvez en savoir plus sur l'indentation dans le contexte des lignes de continuation à https://www.python.org/dev/peps/pep-0008/#indentation.

Longueur maximale de ligne et sauts de ligne

Si possible, vous devez **limiter toutes les lignes à un maximum de 79 caractères**, car cela vous évitera d'envelopper plusieurs lignes de code. Si le retour à la ligne est inévitable, utilisez la continuation de ligne implicite de Python du point précédent.

Dans le cas des docstrings et des commentaires, la longueur de la ligne ne doit pas dépasser 72 caractères.



Pour plus de commodité, si une équipe (ou des équipes) travaillant sur un projet donné, selon PEP 8, il est possible d'augmenter la longueur de la ligne à 99 caractères (cela ne concerne pas les docstrings/commentaires, dont la longueur de ligne doit rester limitée à 72 caractères).
Néanmoins, la bibliothèque standard Python est conservatrice en la matière et vous oblige à ne pas utiliser plus de 79 caractères par ligne (72 pour les commentaires/docstrings).

Sauts de ligne et opérateurs

Même si en Python vous êtes autorisé à casser des lignes de code avant ou après les opérateurs binaires (à condition que vous le fassiez régulièrement et que cette convention ait déjà été utilisée dans votre code), il est recommandé de suivre les suggestions de style de Donald Knuth et de **casser <u>avant</u>** les **opérateurs binaires** car cela se traduit par un code plus lisible et plus convivial.

Exemple:

```
# Recommended

total_fruits = (apples
                + pears
                + grapes
                - (black currants - red currants)
                - bananas
                + oranges)
```

Lignes vides (blanches)

Les lignes vides, appelées *espaces verticaux*, améliorent la lisibilité de votre code.
Ils permettent à la personne qui lit votre code de voir la division du code en sections, Ils l'aident à mieux comprendre la relation entre les sections et à saisir plus facilement la logique de blocs de code donnés.

De la même manière, utiliser trop de lignes vides dans votre code le rendra clairsemé et plus difficile à suivre, c'est pourquoi vous devez toujours faire attention à ne pas en abuser.
Le PEP 8 recommande d'utiliser :

– **deux lignes** blanches pour entourer les définitions de fonction et de classe de niveau supérieur :

```
class ClassOne:
    pass


Class ClassTwo:
    pass


def my_top_level_function():
    return None
```

– une **seule ligne vide** pour entourer les définitions de méthode à l'intérieur d'une classe:

```
class MyClass:
    def method_one(self):
        return None

    def method_two(self):
        return None
```

– lignes **blanches dans les fonctions afin d'indiquer les sections logiques** (avec parcimonie). Par exemple:

```
def calculate_average():
    how_many_numbers = int(input("How many numbers? "))

    if how_many_numbers > 0:
        sum_numbers = 0
        for i in range(0, how_many_numbers):
            number = float(input("Enter a number: "))
            sum_numbers += number

        average = 0
        average = sum_numbers / how_many_numbers

        return average
    else:
        return "Nothing happens."
```

3.5 PEP 8 Codage des fichiers sources et importations de modules

3.5.1 Codages par défaut

Il est recommandé **d'utiliser les codages par défaut de Python (Python 3 -- UTF-8, Python 2 -- ASCII).** Les codages autres que ceux par défaut sont déconseillés et ne doivent être utilisés qu'à des fins de test ou dans des situations où vos commentaires ou docstrings utilisent un nom (par exemple, le nom d'un auteur) qui contient un caractère non-ASCII.
PEP 8 stipule que « tous les identifiants de la bibliothèque standard Python DOIVENT utiliser **des identifiants ASCII uniquement, et DEVRAIENT utiliser des mots anglais chaque fois que possible** ».

Remarque : Voir PEP 3131 (Prise en charge des identificateurs non-ASCII) pour plus d'informations sur la justification ainsi que les avantages et les inconvénients de l'utilisation d'identificateurs non-ASCII.

3.5.2 Importations

Vous devez toujours **placer les importations au début de votre script**, entre les commentaires de module/docstrings et les variables globales et constantes du module, en respectant l'ordre suivant :

1. Importations de bibliothèques standard ;
2. Importations de tiers liés;
3. Importations spécifiques à l'application/bibliothèque locale.

Assurez-vous d'insérer une ligne vide pour séparer chacun des groupes d'importations ci-dessus.

Le PEP 8 recommande que vos importations soient sur **des lignes distinctes**, plutôt que sur une seule ligne :

```
# Bad:

import sys, os
```

```
# Good:

import os
import sys
```

Néanmoins, il est correct de faire une importation d'une ligne en utilisant ***from … import …*** :

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

Si possible, utilisez des importations **absolues** (c'est-à-dire des importations qui utilisent des chemins absolus séparés par des points). Par exemple:

```
import animals.mammals.dogs.puppies
```

De telles importations sont préférées en Python, en particulier lorsque votre application n'est pas envahie par un sur développement ou extrêmement complexe.
Vous ne devriez pas (et en fait vous ne pouvez pas) utiliser des importations relatives implicites, car celles-ci ne sont plus présentes dans Python 3. Vous devez également **éviter d'utiliser des importations génériques**, comme :

```
from animals import *
```

car ils inhibent la lisibilité du code et peuvent interférer avec certains des noms déjà présents dans l'espace de noms.

3.6 PEP 8 – Guillemets chaînes, espaces et virgules de fin

3.6.1 Recommandations pour les guillemets de chaîne, les espaces et les virgules de fin

Dans cette section, nous nous concentrerons sur les recommandations de style liées à des éléments tels que :

- guillemets de chaîne;
- espace blanc dans les expressions et les instructions, et l'utilisation de virgules de fin.

3.6.2 Guillemets de chaîne

Python nous permet d'utiliser des guillemets simples (par exemple, 'une chaîne') et des guillemets doubles (par exemple, « une chaîne »). Ce sont les mêmes informations, et il n'y a pas de recommandation spéciale dans PEP vous indiquant quel style vous devriez adopter dans votre écriture de code. Encore une fois, la règle la plus importante est : soyez cohérent avec votre choix.

Cependant, pour améliorer la lisibilité, PEP 8 recommande **d'essayer d'éviter d'utiliser des barres obliques inverses** (caractères d'échappement) dans les chaînes.
Cela signifie que :

- Si votre chaîne contient des guillemets simples, il est recommandé d'utiliser des chaînes entre guillemets doubles ;
- Si votre chaîne contient des guillemets doubles, il est recommandé d'utiliser des chaînes entre guillemets simples.

Dans le cas des chaînes entre guillemets triples, PEP 8 vous recommande de toujours utiliser des guillemets doubles pour maintenir la cohérence avec la convention docstring détaillée dans PEP 257.

3.6.3 Espace dans les expressions et les instructions

PEP 8 contient une longue section qui montre des exemples d'utilisations correctes et incorrectes des espaces blancs dans le code. En règle générale, vous devez **éviter d'utiliser trop d'espaces**, car cela rend votre code difficile à suivre.

Ainsi, par exemple, **n'utilisez pas** d'espace blanc excessif immédiatement **entre parenthèses/crochets/accolades**, ou **immédiatement avant une virgule/ point-virgule/deux-points**:

```
# Bad:

my_list = ( dog[ 2 ] , 5 , { "year": 1980 } , "string" )
if 5 in my_list : print( "Hello!" ) ; print( "Goodbye!" )
```

```
# Good:

my_list = (dog[2], 5, {"year": 1980}, "string")
if 5 in my_list: print("Hello!"); print("Goodbye!")
```

3.6.4 Espace dans les expressions et les instructions (suite)

Dans le cas d'une tranche, le signe deux-points doit avoir des quantités égales d'espace des deux côtés (il doit agir comme un opérateur binaire) à moins qu'un paramètre de tranche ne soit omis, auquel cas l'espace doit également être omis.

Exemples:

```
# Bad:

bread[0 : 3], roll[1: 3 :5], bun[3: 5:], donut[ 1: :5 ]
```

```
# Good:

bread[0:3], roll[1:3:5], bun[3:5:], donut[1::5]
```

3.6.5 Virgules de fin

Encore une fois, n'utilisez pas trop d'espaces blancs :

- après une virgule de fin suivie d'une parenthèse fermante, ou
- immédiatement avant une parenthèse ouvrante qui marque le début de la liste d'arguments d'un appel de fonction, ou
- immédiatement avant une parenthèse ouvrante qui marque le début de l'indexation/découpage.

```
# Bad:

my_tuple = (0, 1, 2, )
my_function (5)
my_dictionary ['key'] = my_list [index]
```

```
# Good:

my_tuple = (0, 1, 2,)
my_function(5)
my_dictionary['key'] = my_list[index]
```

3.6.6 Espace dans les expressions et les instructions (suite)

N'utilisez pas plus d'un espace avant et après les opérateurs, par exemple :

```
# Bad:

a          = 1
b          = a         + 2
my_string = 'string' * 2
```

```
# Good:

a = 1
b = a + 2
my_string = 'string' * 2
```

Entourez les opérateurs binaires d'un seul espace des deux côtés. Toutefois, si dans votre code il y a des opérateurs qui ont des priorités différentes, vous pouvez envisager d'ajouter un espacement autour des opérateurs de priorité faible (la plus faible) uniquement, par exemple :

```
# Bad:

x=x+3
x -=1

x = x * 2 - 1
x = (x - 1) * (x + 2)
```

```
# Good:

x = x + 3
x -= 1

x = x*2 - 1  # Utilisez votre jugement.
x = (x-1) * (x+2)  # Utilisez votre jugement.
```

N'entourez pas l'opérateur **=** d'espaces s'il est utilisé pour indiquer un argument de mot-clé/valeur par défaut, par exemple :

```
# Bad:

def my_function(x, y = 2):
    return x * y
```

```
# Good:

def my_function(x, y=2):
    return x * y
```

3.7 PEP 8 – Commentaires

3.7.1 Recommandations pour l'utilisation des commentaires

Les commentaires sont destinés à améliorer la lisibilité du code sans affecter la sortie du programme. Les bons programmeurs documentent leur code et expliquent les extraits de code les plus complexes, de sorte que la personne qui lit le code comprenne correctement ce qui se passe dans le programme. Vous devez utiliser les commentaires à bon escient et, dans la mesure du possible, écrire du code qui s'auto-commentera (par exemple, donnez des noms propres à vos variables, fonctions et éléments de code). Certaines règles tournent vers le « no comment » mais alors le code doit être très lisible.

Il y a quelques règles que vous devez suivre lorsque vous laissez des commentaires dans le code :

- Rédigez des commentaires qui ne contrediront pas le code ou n'induiront pas le lecteur en erreur. Ils sont bien pires que pas de commentaire du tout.
- Mettez à jour vos commentaires lorsque votre programme est mis à jour.
- Écrivez des commentaires sous forme de **phrases complètes** (mettez en majuscule le premier mot s'il ne s'agit pas d'un identificateur et terminez votre phrase par un point).

Par exemple:

```
# Program that calculates body mass index (BMI).

height = float(input("Your height (in meters): "))
weight = float(input("Your weight (in kilograms): "))
bmi = round(weight / (height*height), 2)

print("Your BMI: {}".format(bmi))
```

- Lorsque vous écrivez des commentaires de bloc avec des commentaires de plusieurs phrases, utilisez deux espaces après chaque point de fin de phrase, sauf après la dernière phrase.
- Écrivez des commentaires en anglais (sauf si vous êtes sûr à 100% que le code ne sera jamais lu par des personnes qui ne parlent pas votre langue.)
- Les commentaires ne doivent pas comporter plus de 72 caractères par ligne (mais vous le savez déjà).

3.7.2 Les commentaires en bloc

Les commentaires sont généralement plus longs et vous devez les utiliser pour **expliquer des sections de code** plutôt que des lignes particulières. Ils vous permettent de laisser des informations pour le lecteur en plusieurs lignes (et plusieurs phrases). En règle générale, bloquez les commentaires :

- devraient se référer au code qui les suit;
- doivent être mis en retrait au même niveau que le code qu'ils décrivent.

Lorsque vous écrivez des commentaires de bloc, commencez chaque ligne par # suivi d'un seul espace et séparez les paragraphes par une ligne contenant uniquement le symbole #. Par exemple:

```
def calculate_product():
    # Calculate the average of three numbers obtained from the
user. Then
    # multiply the result by 4.17, and assign it to the
product variable.
    #
    # Return the value passed to the product variable and use
it
    # for the subsequent x to y calculations to speed up the
process.
    sum_numbers = 0

    for number in range(0, 3):
        number = float(input("Enter a number: "))
        sum_numbers += number

    average = (sum_numbers / 3) * 4.17
    product = average
    return product

x = product * 1.73
y = x ** 2
x to v = (x*v) / 1.05
```

3.7.3 Commentaires en ligne

Les commentaires en ligne sont des commentaires écrits **sur la même ligne que vos relevés**. Ils devraient aborder ou fournir **des explications supplémentaires sur une seule ligne de code ou une seule déclaration**. Vous ne devriez pas en abuser.

En règle générale, les commentaires en ligne doivent être :

- séparés par deux espaces (ou plus) de l'énoncé auquel ils s'adressent ;
- utilisé avec parcimonie.

Ils peuvent rapidement vous aider à vous souvenir de ce que fait une ligne de code particulière, ou être utiles lorsqu'ils sont lus par quelqu'un qui ne connaît pas votre code. Par exemple:

```
counter = 0      # Initialize the counter.
```

Cependant, n'utilisez pas de commentaires en ligne (ou tout autre commentaire!) pour expliquer des choses évidentes ou inutiles. Par exemple:

```
a += 1     # Increment a.
```

Essayez toujours de faire en sorte que votre code s'auto-commente plutôt que d'ajouter des commentaires, même s'ils semblent raisonnables ou nécessaires, par exemple :

```
# Bad:

a = 'Adam'  # User's first name.
```

```
# Good:

user_first_name = 'Adam'
```

3.7.4 Chaînes de documentation

Les chaînes de documentation, ou **docstrings** comme on les appelle souvent, vous permettent de fournir des **descriptions et des explications pour tous les modules, fichiers, fonctions, classes et méthodes publics** que vous utilisez dans votre code. Vous devez les utiliser dans ce contexte.

Nous allons traiter des docstrings lorsque nous parlerons de **PEP 257** plus tard dans le cours. Pour le moment, il suffit de se rappeler qu'il s'agit d'un type de commentaire qui commence et se termine par trois guillemets doubles : « ». Exemples:

```
# A multi-line docstring:

def fun(x, y):
    """Convert x and y to strings,
    and return a list of strings.
    """
    ...
```

```
# A single-line docstring:

def fun(x):
    """Return the square root of x."""
    ...
```

3.8 PEP 8 – Conventions d'appellation

3.8.1 Conventions d'appellation – Introduction

Lors de votre programmation, vous devez souvent nommer des identifiants et d'autres entités dans votre code. Donner des noms appropriés et éviter les noms inappropriés augmentera la lisibilité de votre code et vous fera économiser (ainsi qu'aux autres programmeurs qui lisent votre code) beaucoup de temps et d'efforts.

Vous suivrez dans votre carrière certainement différentes conventions pour donner des noms aux variables, fonctions et classes dans votre code ; Certaines d'entre elles peuvent provenir de vos anciennes études dans d'autres langages, d'autres peuvent être un choix purement pratique, tandis que d'autres encore peuvent être déterminés par les exigences du projet ou les pratiques adoptées par votre entreprise ou votre équipe.

Les conventions de nommage Python ne sont malheureusement pas entièrement cohérentes dans l'ensemble de la bibliothèque Python. Cependant, il est recommandé que les nouveaux modules et packages soient écrits conformément aux recommandations de nommage du PEP 8 (sauf si une bibliothèque existante suit un style différent, auquel cas la cohérence interne est la solution préférée).

3.8.2 Dénomination des styles

Il existe de nombreux styles de nommage différents utilisés en programmation, par exemple:

`a` – lettre minuscule unique
`A` – lettre majuscule simple

En règle générale, vous devriez éviter d'utiliser des noms à une seule lettre comme l (la lettre minuscule L), I (la lettre majuscule de i) et O (la lettre majuscule de o), car ils peuvent facilement être confondus avec les chiffres 1 et 0, et rendre votre code beaucoup moins lisible.

- **mynamplename**: minuscule
- **my_sample_name**: minuscules avec traits de soulignement(snake_case)
- **MYSAMPLENAME** : majuscules
- **MY_SAMPLE_NAME** : majuscules avec traits de soulignement (SNAKE_CASE)

- **MySampleName** : CamelCase (également connu sous le nom de mots en majuscules, StudlyCaps ou CapWords)

  Une petite note: lorsque vous utilisez des acronymes, vous devez mettre en majuscule toutes les lettres qui composent l'acronyme, par exemple, **HTTPServerError**

- **mySampleName** : casse mixte, qui ne diffère en fait de CamelCase que par un caractère minuscule initial

- **My_Sample_Name** : mots en majuscules avec traits de soulignement (considérés comme laids par PEP 8)

- **_my_sample_name :** un nom qui commence par un seul trait de soulignement indique une faible « utilisation interne », par exemple, l'instruction : **from SAMPLE import *** n'importera pas les objets dont le nom commence par un trait de soulignement.

- **my_sample_name_** : un seul trait de soulignement de fin est utilisé par convention afin d'éviter tout conflit avec les mots-clés Python, par exemple, class_

- **__my_sample_name** : un nom qui commence par un trait de soulignement double est utilisé pour les attributs de classe où il appelle la manipulation de nom, par exemple, à l'intérieur de la classe MySampleClass, __room deviendra **_MySampleClass__room**

- **__my_sample_name__** : Un nom qui commence et se termine par un double trait de soulignement est utilisé pour les objets et attributs « magiques » qui résident dans des espaces de noms contrôlés par l'utilisateur, par exemple, __init__, __import__ ou __file__. Vous ne devez pas créer de tels noms, mais uniquement les utiliser tels que documentés.

3.8.3 Conventions d'appellation – recommandations

Le PEP 8 prévoit une convention d'appellation spécifique en ce qui concerne un identificateur spécifique.

Lorsque vous donnez un nom à une **variable**, vous devez utiliser une lettre minuscule ou un ou plusieurs mots, et séparer les mots par des traits de soulignement, par exemple, x, var, my_variable.
La même convention s'applique aux variables globales.

Les fonctions suivent les mêmes règles que les variables, c'est-à-dire que lorsque vous donnez un nom à une **fonction**, vous devez utiliser une lettre minuscule ou un ou plusieurs mots séparés par des traits de soulignement, par exemple, fun, my_function.

Lorsque vous donnez un nom à une **classe**, vous devez adopter le style CamelCase, par exemple, MySampleClass, ou s'il n'y a qu'un seul mot, commencer par une majuscule, par exemple, Sample.

Lorsque vous donnez un nom à une **méthode**, vous devez utiliser un mot minuscule ou des mots séparés par des traits de soulignement, par exemple, method, my_class_method.
Vous devez toujours utiliser **self** comme premier argument des méthodes d'instance et **cls** pour le premier argument des méthodes de classe.

Lorsque vous donnez un nom à une **constante**, vous devez utiliser des lettres majuscules et séparer les mots par des traits de soulignement, par exemple, TOTAL, MY_CONSTANT.

Lorsque vous donnez un nom à un **module**, vous devez utiliser un ou plusieurs mots minuscules, de préférence courts, et les séparer par des traits de soulignement, par exemple, samples.py, my_samples..

Lorsque vous donnez un nom à un **package**, vous devez utiliser un ou plusieurs mots minuscules, de préférence courts. Vous ne devriez pas séparer les mots, par exemple, package, mypackage.

Les noms de variables de type doivent suivre la convention CamelCase et être courts, par exemple, AnyStr ou Num.

Lorsque vous donnez un nom à une **exception**, vous devez suivre la même convention qu'avec les classes (gardez à l'esprit que les exceptions doivent en fait être des classes), c'est-à-dire utiliser le style CamelCase.

Remarque : Vous pouvez utiliser un style différent, par exemple, casse mixte (mySample) pour les fonctions et les variables, mais uniquement si cela permet de conserver la rétrocompatibilité, et si c'est le style dominant.

Pour plus d'informations sur les conventions de nommage PEP 8, rendez-vous sur la page officielle du PEP 8 : https://www.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions.

3.9 PEP 8 – Recommandations de programmation

3.9.1 Recommendations de programmation

Il existe souvent plusieurs façons d'écrire du code qui effectuera la même action en Python, mais PEP 8, encore une fois, impose certaines conventions et fournit des conseils sur la façon dont vous devez suivre les meilleures pratiques de programmation pour éviter toute ambiguïté, maintenir la cohérence avec votre code précédent et vos bibliothèques Python et obtenir de meilleures performances / efficacité du code.
Les voilà:

– faire des comparaisons avec l'objet **None** quand vous utilisez **is** ou **is not**, mais pas avec les opérateurs d'(in)égalité (== et !=), par exemple :

```
# Bad:

if x == None:
    print("A")
```

```
# Good:

if x is None:
    print("A")
```

– N'utilisez pas les opérateurs d'(in)égalité lorsque vous comparez des valeurs booléennes à True ou False. Encore une fois, l'utilisation de **is** ou **is not** à la place:

```
# Bad:

my_boolean_value = 2 > 1
if my_boolean_value == True:
    print("A")
else:
    print("B")
```

```
# Good:

my_boolean_value = 2 > 1
if my_boolean_value is True:
    print("A")
else:
    print("B")
```

```
# Better:

my_boolean_value = 2 > 1
if my_boolean_value:
    print("A")
else:
    print("B")
```

– Pour des raisons de lisibilité, utilisez l'opérateur **is not** et pas **not...is** :

```
# Bad:

if not x is None:
    print("It exists")
```

```
# Good:

if x is not None:
    print("It exists")
```

Remarque : évitez d'utiliser **if x:** pour exprimer si **x is not None** : lorsque vous souhaitez vérifier si une variable ou un argument donné est défini sur None par défaut et s'est vu attribuer une valeur différente.

– lorsque vous souhaitez « attraper » une exception, référez-vous à des exceptions spécifiques plutôt que d'utiliser la clause except uniquement:

```
try:
    import my_module
except ImportError:
    my_module = None
```

– Lorsque vous recherchez des préfixes ou des suffixes, utilisez les méthodes de chaîne ''.startswith() et ''.endswith(), car elles sont plus propres et moins sujettes aux erreurs. En règle générale, il est préférable d'utiliser des méthodes de chaîne plutôt que d'importer le module de string.

```
# Bad:

if name[:4] == 'Adam':
    # do something
```

```
# Good:

if name.startswith('Adam'):
    # do something
```

Pour plus de suggestions sur la façon d'écrire un meilleur code et les pratiques à éviter, consultez la page officielle des recommandations de programmation PEP 8 : https://peps.python.org/pep-0008/#programming-recommendations .

4. Qu'est-ce que le PEP 257?

PEP 257 est un document créé dans le cadre du Guide du développeur Python, qui tente de normaliser la structure de haut niveau des docstrings. Il décrit les **conventions**, les meilleures pratiques et la sémantique (et non pas les lois ou les règlements !) associés à la documentation du code Python à l'aide de docstrings. En bref, il tente de répondre aux deux questions suivantes:

- **Que** doivent contenir les docstrings Python ?
- **Comment** utiliser les docstrings Python ?

4.1 Que sont les docstrings ?

Un docstring est un littéral de chaîne qui apparaît comme la première instruction d'une définition de module, de fonction, de classe ou de méthode.
Un tel docstring commence par **__doc__** qui est un attribut spécial de cet objet.

En d'autres termes, les **docstrings** sont des chaînes de documentation Python utilisées dans la définition de classe, de module, de fonction et de méthode afin de fournir des informations sur les fonctionnalités d'un morceau de code plus volumineux de manière à le mettre en **prescriptive**.

Ils aident les programmeurs (y compris vous) à se souvenir et à comprendre le but, le fonctionnement et les capacités de blocs de code ou de sections particuliers.

4.2 Docstring et commentaires

Avant de poursuivre, nous devons comprendre cette distinction essentielle (comme leur nom l'indique) : les commentaires sont utilisés pour **commenter** votre code , tandis que les docstrings sont utilisés pour **documenter** votre code. Alors, quelle est la différence entre les commentaires et les docstrings, et éventuellement entre le commentaire et la documentation du code ?

Regardez le tableau à la page suivante, où nous voulons vous montrer certaines des différences entre les commentaires et les docstrings en Python:

Commentaires	Docstrings
Les commentaires sont des instructions non exécutables en Python, ce qui signifie qu'ils sont ignorés par l'interpréteur Python ; Ils ne sont pas stockés dans la mémoire et ne sont pas accessibles pendant l'exécution du programme (c'est-à-dire qu'ils sont accessibles en regardant le code source).	Les docstrings sont accessibles en lisant le code source et en utilisant l'attribut __doc__ ou la fonction help().
L'objectif principal des commentaires est d'augmenter la lisibilité et la compréhensibilité du code et d'expliquer le code à l'utilisateur de manière significative. L'utilisateur ici signifie à la fois les autres programmeurs et vous (par exemple, lorsque vous revenez à votre code après un certain temps) : quelqu'un qui voudra ou aura besoin de modifier, étendre ou maintenir le code.	L'objectif principal de docstrings est de documenter votre code : en décrivant son utilisation, ses fonctionnalités et ses capacités aux utilisateurs qui n'ont pas nécessairement besoin de savoir comment il fonctionne.
Les commentaires ne peuvent pas être transformés en documentation; Leur but est de simplifier le code, de fournir des informations précises et d'aider à comprendre l'intention d'un extrait / ligne particulier.	Les docstrings peuvent être facilement transformés en documentation réelle, qui décrit le comportement d'un module ou d'une fonction, la signification des paramètres ou le but d'un package spécifique.

Bien sûr, comme vous le verrez dans les pages suivantes, il y a beaucoup plus à dire sur les docstrings: comment les utiliser, pourquoi les utiliser et où; Et (comme vous vous en doutez) la différence entre les commentaires et les docstrings deviendra encore plus évidente pour vous.

4.3 Pourquoi commenter? Pourquoi documenter ?

Avant d'entrer dans le sujet des docstrings, essayons de répondre à la question : pourquoi est-il important de commenter et de documenter le code ?

Nous ne devons pas oublier cette règle simple de Guido van Rossum : « **Le code est plus souvent lu qu'écrit** », ce qui signifie essentiellement que le code que nous écrivons aujourd'hui sera très probablement lu à l'avenir : soit par vous, soit par un autre programmeur, soit même par des équipes de programmeurs.

Il est donc crucial que nous développions de telles habitudes de programmation et d'écriture de code qui permettront aux développeurs et aux autres utilisateurs de comprendre le pourquoi et le comment de ce code, cela facilitera grandement la réutilisation et la contribution au code.

Nous devrions donc convenir que la documentation du code aide à maintenir un code plus propre, plus lisible et plus durable, ce qui signifie que c'est l'une des meilleures pratiques qu'un bon développeur responsable devrait adopter dans le cadre de son ensemble d'outils d'aide à la programmation.

4.4 Un rapide récapitulatif des commentaires

Nous espérons que vous vous souviendrez que les commentaires en Python sont créés à l'aide du signe dièse (#). Ils doivent être assez brefs (pas plus de 72 caractères par ligne), commencer par une majuscule et se terminer par un point. Si vous devez inclure un commentaire plus long dans votre code, vous pouvez utiliser un commentaire de plusieurs lignes, auquel cas vous devez utiliser le signe dièse au début de chaque ligne de commentaire.

En règle générale, vous devez insérer **des commentaires proches** du code que vous décrivez afin de préciser au lecteur à quelle partie du code vous faites référence. Vous devez être précis : n'incluez pas d'informations non pertinentes ou redondantes; Et surtout, essayez de concevoir et d'écrire votre code de manière à ce qu'il se commente facilement et de manière compréhensible (par exemple, donnez des noms auto-commentatés aux variables).

4.5 Quand utiliser les commentaires ?

Outre les cas les plus évidents, tels que les **descriptions de code et d'algorithmes**, les commentaires peuvent servir à quelques autres fins utiles. Par exemple :

- Ils peuvent vous aider à **baliser** les sections de code qui doivent être faites à l'avenir, ou qui sont laissées pour une amélioration ultérieure, par exemple :

  ```
  # TODO : Ajoutez une fonction qui prend les arguments val et sum .
  ```

- Ils peuvent vous aider à commenter (et à décommenter) les sections de code que vous souhaitez tester, par exemple :

  ```
  def fun(val):
      return val * 2

  user_value = int(input("Enter the value: "))
  # fun(user_value)
  # user_value = user_value + "foo"

  print(fun(user_value))
  ```

- Ils peuvent vous aider à planifier votre travail et à décrire certaines sections de code que vous allez concevoir, par exemple :

  ```
  # Étape 1: Demandez à l'utilisateur la valeur.
  # Étape 2: Changez la valeur en int et gérez les exceptions possibles.
  # Étape 3: Imprimez la valeur multipliée par 0,7.
  ```

4.6 PEP 484 – Indication de type

4.6.1 Quelques mots sur les conseils de type : PEP 484

Avant de commence à parler de commentaires à élaborer sur les docstrings, il y a une autre fonctionnalité Python dont nous voulons vous parler brièvement : **type hinting**.

Le type hinting est un mécanisme introduit avec Python 3.5 et décrit dans PEP 484 qui vous permet d'équiper votre code d'informations supplémentaires sans utiliser de commentaires. Il s'agit d'une fonctionnalité facultative, mais plus formalisée, qui vous permet d'utiliser le module de **typage intégré** de Python pour fournir des informations d'astuce de type dans votre code afin de laisser certaines suggestions, de marquer certains problèmes possibles qui peuvent survenir dans le processus de développement et d'étiqueter des noms spécifiques avec **des informations** de *type*.

En un mot, le type hinting vous permet d'indiquer **statiquement** les informations de type liées aux objets Python, ce qui signifie que vous pouvez, par exemple, ajouter des informations de type à une fonction : indiquer le type d'un argument accepté par la fonction ou le type de la valeur qu'elle retournera.

Regardez les exemples suivants :

```
# No type information added:
def hello(name):
    return "Hello, " + name


# Type information added to a function:
def hello(name: str) -> str:
    return "Hello, " + name
```

L'indication de type est **facultative**, ce qui signifie que PEP 484 ne vous oblige pas à laisser des informations liées au typage statique dans votre code. Le premier exemple est exempt de tout indice de type.

Dans le deuxième exemple, l'annotation str, qui indique que l'argument *name* passé à la fonction *hello()* doit être de type *str*, nous aide à minimiser le risque de certaines situations (in)attendues : elle réduit le risque de transmettre un type de valeur incorrect à la fonction. L'annotation -> *str* indique également que la fonction *hello()* retournera une valeur de type *str*, qui est bien sûr une chaîne.

Qu'est-ce que tout cela signifie pour vous, et comment pouvez-vous tirer parti des indices de type en Python?

- Tout d'abord, les type hinting peuvent vous aider à **documenter votre code**. Au lieu de laisser les informations relatives aux arguments et aux réponses dans docstrings, vous pouvez utiliser le langage lui-même à cette fin. Cela peut être un moyen élégant et utile de mettre en évidence certaines des informations de code les plus importantes, en particulier lorsque vous publiez du code dans un projet, le partagez avec d'autres développeurs ou laissez des conseils pour vous-même lorsque vous devez revenir au code source à l'avenir. Dans certains des plus grands projets de développement logiciel, l'indication de type est une pratique recommandée qui aide les équipes à mieux comprendre la façon dont les types s'exécutent dans le code.

- Les type hinting vous permettent **de remarquer certains types d'erreurs plus efficacement** et d'écrire un code plus beau et, surtout, plus **propre**. Lorsque vous utilisez des type hinting, vous réfléchissez plus attentivement aux types dans votre code, ce qui permet d'éviter ou de détecter certaines des erreurs pouvant résulter de la nature dynamique de Python. (Cependant, nous ne préconisons pas que Python nécessite un typage statique.)

- Vous devez vous rappeler que les type hinting en Python **ne sont pas utilisés** lors de l'exécution, ce qui signifie que toutes les informations de type que vous laissez dans le code sous forme d'annotations sont effacées lors de l'exécution du programme. En d'autres termes, l'indication de type n'a aucun effet sur le fonctionnement de votre code. D'autre part, lorsqu'il est utilisé avec un système de vérification de type ou des lint-like(outils de vérification statique de code) que vous pouvez brancher à votre éditeur ou IDE, il peut prendre en charge votre écriture de code en le complétant automatiquement celui-ci, également en repérant et en mettant en évidence les erreurs avant que votre code ne soit exécuté.

- Étant donné que les type hinting n'ont aucun effet sur le code source, cela signifie qu'ils n'ont aucun impact sur les temps de performance (les caractères sont ignorés par Python au moment de l'exécution, ce qui n'a aucune influence sur les accélérations d'interprétation /compilation).

4.7 PEP 257 – Conventions de docstring

4.7.1 Docstrings : où et comment ?

Nous avons déjà dit que les docstrings peuvent être utilisés dans les classes, les modules, les fonctions et les définitions de méthodes. Nous voulons maintenant compléter ce point: il y a des cas où non seulement ***ils peuvent*** être inclus, mais devraient l'être. Pour être plus précis, tous les modules, fonctions, classes et méthodes publics exportés par un module donné **doivent avoir des docstrings**.

Les méthodes non publiques n'ont pas besoin de contenir des docstrings. Cependant, il est recommandé de laisser un commentaire juste après la ligne **def** décrivant ce que la méthode fait réellement. Dans le cas des **packages**, ceux-ci doivent également être documentés, et vous pouvez écrire des docstrings de package dans le module docstring du fichier **__init__.py** dans le dossier package.

Comme nous l'avons déjà dit, les docstrings sont des littéraux de chaîne qui apparaissent comme **la première instruction** d'un module, d'une fonction, d'une classe ou d'une méthode.
Cependant, il est important (et juste) d'ajouter que les littéraux de chaîne peuvent également apparaître dans de nombreux autres endroits du code, et servent toujours de documentation.
Et même s'ils ne sont plus accessibles en tant qu'attributs d'objet d'exécution, ils peuvent toujours être extraits par certains outils logiciels spécifiques.

Et maintenant, sans entrer dans trop de détails, il suffit de vous dire que nous distinguons deux types de « dosctrings supplémentaires », ce sont:

- **DocStrings d'attribut**, qui se trouvent immédiatement après une instruction d'affectation au niveau supérieur d'un module (attributs de module), d'une classe (attributs de classe) ou de la définition de méthode __init__ d'une classe (attributs d'instance). Ceux-ci sont interprétés par les outils d'extraction, tels que Docutils, comme « les docstrings de la cible de l'instruction assignment ». (Si vous souhaitez en savoir plus sur les docstrings attributaires, vous êtes bienvenu dans PEP 224. À ce stade, nous voulons simplement que vous en soyez conscient.)

- **Dosctrings supplémentaires**, qui sont situés immédiatement après un autre docstring. (L'idée originale de docstrings supplémentaires a été tirée du PEP 216, qui à son tour a été remplacé par PEP 287 : encore une fois, vous pouvez jeter un coup d'œil.)

4.7.2 Comment créer des docstrings

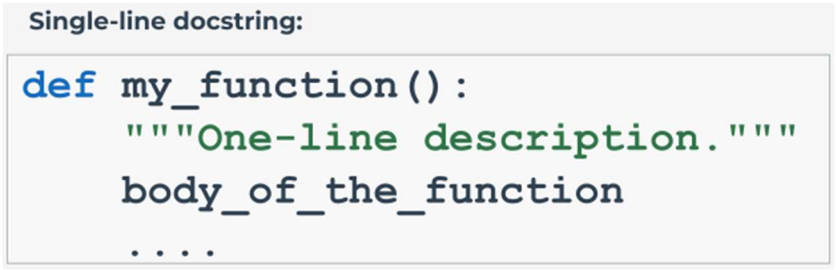
Les docstrings doivent être entourés de triple guillemets doubles (**« « « triple guillemets doubles » » »**). Par exemple:

```
def my_function():
    """I am a docstring."""
    ...
```

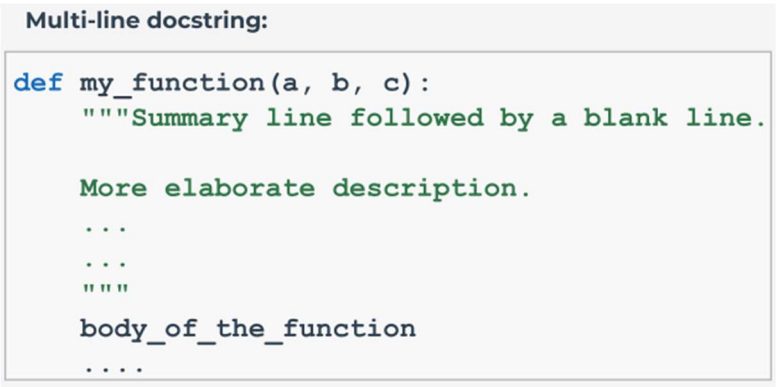
4.7.3 Chaînes de documents d'une ligne et de plusieurs lignes

Il existe deux formes de docstrings. Et même si chacune d'elles sert le même but (c'est-à-dire est censé fournir de la documentation), celui que vous allez utiliser dépendra de certaines conditions comme la quantité d'informations qu'il est nécessaire de fournir. Il s'agit de :

- **Docstrings d'une ligne** : ils sont utilisés pour les descriptions simples et courtes, et doivent tenir sur une seule ligne;



- **DocStrings multi-lignes** : Ils sont utilisés pour les cas plus difficiles et doivent consister en une ligne de résumé suivie d'une ligne vide et d'une description plus élaborée.



4.7.4 Chaînes de documents d'une ligne

Les docstrings d'une ligne doivent être utilisés pour des descriptions plutôt simples, évidentes et courtes. Ils ne doivent occuper qu'une seule ligne comme le nom l'indique et être entourés de triples guillemets doubles (les guillemets fermants doivent être sur la même ligne que les guillemets d'ouverture car cela aide à garder le code propre et élégant).

Remarques importantes :

- Un DocString doit commencer par une lettre majuscule (sauf si un identificateur commence la phrase) et se terminer par un point ;
- Un DocString doit ***prescrire*** l'effet du segment de code, **pas** le décrire. En d'autres termes, il devrait prendre la forme d'un impératif (par exemple, « Faire ceci », « Retourner cela », « Calculer ceci », « Convertir cela », etc.), et non une description (par exemple, « Fait ceci », « Retourne cela », « Forme ceci », « Étend cela », etc.).
  Par exemple:

```
def Salutation(name):
      """Prendre un nom et envoyez sa forme repliqué."""
return name * 2
```

- Un DocString **ne doit pas** simplement répéter les paramètres de la fonction ou de la méthode. Par exemple:

```
def my_function(x, y):
    """my_function(x, y) -> list"""
...
```

Au lieu de cela, essayez de faire quelque chose comme ceci:

```
def my_function(x, y):
    """Compute the angles and return a list of coordinates."""
...
```

- N'utilisez pas de ligne vide au-dessus ou sous un docstring d'une ligne, sauf si vous documentez une classe, auquel cas vous devez placer une ligne vide après toutes les docstrings qui la documentent :

```
def calculate_tax(x, y):

    """I am a one-line docstring."""

    return (x+y) * 0.25
```

```
def calculate_tax(x, y):
    """I am a one-line docstring."""
    return (x+y) * 0.25
```

4.7.5 Chaînes de documents multilignes

Les docstrings multilignes doivent être utilisés pour les cas non évidents et les descriptions plus détaillées des segments de code. Ils doivent avoir une ligne de résumé, similaire à ce à quoi ressemble un docstring d'une ligne, suivie d'une **ligne vide** et d'une description plus élaborée. La ligne de résumé peut être située sur la même ligne que les triples guillemets doubles ouverts, ou placée sur la ligne suivante. Les guillemets de fin doivent être placés sur une ligne séparée.

Remarques importantes :

- Un DocString multiligne doit être mis en retrait au même niveau que les guillemets ouverts, par exemple :

```
def king_creator(name="Johan", ordinal="I", country="Neverland"):
    """Create a king following the article title naming convention.

    Keyword arguments:
    :arg name: the king's name (default: Johan)
    :type name: str
    :arg ordinal: Roman ordinal number (default: I)
    :type ordinal: str
    :arg country: the country ruled (default: Neverland)
    :type country: str
    """
    if name == "Palpatine":
        return "Palpatine is a reserved name."
    ...
```

- Vous devez insérer une ligne vide après toutes les DocStrings multilignes qui documentent une classe ;

- Les docstrings de script (dans le sens programmes autonomes/exécutables à fichier unique) doivent documenter la fonction du script, la syntaxe de ligne de commande, les variables d'environnement et les fichiers. La description doit être équilibrée de manière à aider les nouveaux utilisateurs à comprendre l'utilisation du script, ainsi qu'à fournir une référence rapide à toutes les fonctionnalités du programme pour l'utilisateur plus expérimenté;

- Les docstrings du module doivent répertorier les classes, exceptions et fonctions exportées par le module ;

- DocStrings du package (entendu comme le DocString du module __init__.py du package) doit répertorier les modules et sous-packages exportés par le package ;

- Les docstrings pour les fonctions et les méthodes de classe doivent résumer leur comportement et fournir des informations sur les arguments (y compris les arguments facultatifs), les valeurs, les exceptions, les restrictions, etc.

- Les docstrings de classe doivent également résumer son comportement et documenter les méthodes publiques et les variables d'instance. Par exemple:

```
class Vehicle:
    """A class to represent a Vehicle.

    Attributes:
    -----------
    vehicle_type: str
        The type of the vehicle, e.g. a car.
    id_number: int
        The vehicle identification number.
    is_autonomous: bool
        self-driving -> True, not self-driving -> False


    Methods:
    --------
    report_location(lon=45.00, lat=90.00)
        Print the vehicle id number and its current location.
        (default longitude=45.00, default latitude=90.00)
    """

    def __init__(self, vehicle_type, id_number, is_autonomous=True):
        """
        Parameters:
        -----------
        vehicle_type: str
            The type of the vehicle, e.g. a car.
        id_number: int
            The vehicle identification number.
        is_autonomous: bool, optional
            self-driving -> True (default), not self-driving -> False
        """

        self.vehicle_type = vehicle_type
        self.id_number = id_number
        self.is_autonomous = is_autonomous

    def report_location(self, id_number, lon=45.00, lat=90.00):
        """
        Print the vehicle id number and its current location.

        Parameters:
        -----------
        id_number: int
            The vehicle identification number.
        lon: float, optional
            The vehicle's current longitude (default is 45.00)
        lat: float, optional
            The vehicle's current latitude (default is 90.00)
        """

    ...
    ...
```

4.7.6 Types de mise en forme Docstring

Vous avez peut-être remarqué que nous avons utilisé deux formats de docstring différents pour documenter la fonction **king_creator()** et la classe **Vehicle**. Le premier type de formatage est appelé **reStructuredText**, et c'est la norme officielle de documentation Python expliquée et décrite dans PEP 287. Le deuxième exemple utilise le format **docstrings NumPy/SciPy** (qui est une combinaison du format Google docstrings et du format reStructuredText).

Les deux types de formatage sont bons pour créer une documentation formelle, et les deux sont pris en charge par exemple par Sphinx, l'un des générateurs de documentation Python les plus populaires.

Sphinx est un excellent outil pour créer de la documentation pour les projets de développement de logiciels. Il utilise reStructuredText comme langage de balisage et possède de nombreuses fonctionnalités utiles, telles que la prise en charge du format de sortie HTML, le test automatique des extraits de code, des références croisées étendues et une structure hiérarchique, qui vous permet de définir une arborescence de documents.

4.8 Normes de documentation et linters

4.8.1 Comment documenter un projet

Lors de la documentation d'un projet Python, en fonction de la nature du projet (c'est-à-dire privé, partagé, public, open source / domaine public), vous devez avant tout définir ses utilisateurs et réfléchir à leurs besoins. La création **d'un personnage utilisateur** peut être utile ici car elle vous aidera à identifier les façons dont les utilisateurs utiliseront votre projet.

Cela signifie que vous pouvez facilement améliorer leur expérience en réfléchissant à la façon dont ils vont utiliser votre code et en essayant de prédire les problèmes les plus courants qu'ils peuvent rencontrer.

En règle générale, un projet doit contenir les éléments de documentation suivants :

- un **fichier Lisez-moi**, qui fournit un bref résumé du projet, de son objectif et éventuellement de quelques directives d'installation;
- un fichier **examples.py**, qui est un script qui montre quelques exemples d'utilisation du projet ;
- une **licence** sous la forme d'un fichier txt (particulièrement important pour les projets Open Source et Public Domain)
- Un fichier **How to Contribute** qui fournit des informations sur les moyens possibles de contribuer au projet (projets partagés, open source et du domaine public).

Parce que la documentation de votre code peut être une activité plutôt épuisante et chronophage, vous êtes encouragé à utiliser certains outils qui pourraient vous aider à générer automatiquement de la documentation dans le format souhaité et à gérer les mises à jour de la documentation et le contrôle de version de manière efficace et efficiente.

Il existe de nombreux outils et ressources de documentation disponibles, tels que *Sphinx*, que nous avons déjà mentionné, ou le très populaire *pdoc*, et bien d'autres.

4.8.2 Linters et correcteurs

Comment maintenez-vous la bonne qualité de votre code ? Eh bien, vous savez déjà que vous pouvez suivre les guides de style tels que PEP 8 ou PEP 257, et écrire votre code de manière lisible et cohérente.
Vous pouvez (et peut-être devriez) adopter la philosophie Zen of Python, avec tous ses bons conseils pour écrire un code élégant et maintenable, et utiliser le mécanisme d'indication de type.
Vous pouvez observer comment les autres écrivent du code et le documentent dans le cadre de leurs projets (regardez la bibliothèque standard Python ou la bibliothèque Requests), et apprendre d'eux.
Enfin, vous pouvez utiliser *des linters*.

Mais qu'est-ce **qu'un linter** ? Eh bien, c'est un outil qui vous aide à écrire votre code, car il l'analyse **pour détecter toute anomalie stylistique et erreur de programmation par rapport à un ensemble de règles prédéfinies**.
En d'autres termes, il s'agit d'un programme qui analyse votre code et signale les problèmes tels que des erreurs structurelles et syntaxiques, des ruptures de cohérence et un manque de compatibilité avec les meilleures pratiques ou les directives de style de code telles que PEP 8.
Les linters les plus populaires sont: Flake8, Pylint, Pyflakes, Pychecker, Mypy et Pycodestyle (anciennement Pep8) : l'outil linter officiel pour vérifier le code Python par rapport aux conventions PEP 8.

Un **correcteur**, d'autre part, est un programme qui vous aide à *résoudre* des problèmes et à formater votre code pour qu'il soit cohérent avec les normes adoptées. Les correcteurs les plus populaires sont: Black, YAPF et autopep8.

La plupart des éditeurs et des IDE (par exemple, PyCharm, Spyder, Atom, Sublime Text, Visual Studio, Eclipse + PyDev, VIM) prennent en charge les linters, ce qui signifie que vous pouvez les exécuter en arrière-plan lorsque vous écrivez du code. Cela permet de détecter, de mettre en évidence et d'identifier de nombreux problèmes dans votre code, tels que les fautes de frappe, les problèmes de tabulation et d'indentation incorrectes, les appels de fonction avec le mauvais nombre d'arguments, les incohérences stylistiques, les modèles de code dangereux, etc., et de formater automatiquement votre code selon une spécification prédéfinie.

Cela étant dit, nous vous encourageons à explorer vous-même le territoire des linters et des correcteurs et à commencer à les utiliser pour maintenir un code Python de haute qualité et simplement vous faciliter la vie.

4.9 Accès aux docstrings

4.9.1 Comment accéder aux docstrings

La dernière question qui reste en suspens est la suivante: comment pouvons-nous réellement accéder aux docstrings?

Nous le faisons en utilisant l'attribut Python **__doc__** : si des littéraux de chaîne sont présents après la définition d'une fonction / module / classe / méthode, ils sont associés à l'objet en tant qu'attribut __doc__, et cet attribut fournit la documentation de cet objet.

Exécutez le code suivant pour voir ce qui se passe.

```
def my_fun(a, b):
    """ La ligne de résumé va ici.

     Une description plus élaborée de la fonction.

     Paramètres:
     A : Int (description)
     B: int (description)

     Retourne:
     int : Description de la valeur renvoyée.
    """
    return a*b

print(my_fun.__doc__)
```

Votre sortie devrait ressembler à ceci :

La ligne de résumé va ici.

Une description plus élaborée de la fonction.

Paramètres:
A : Int (description)
B: int (description)

Retourne:
int : Description de la valeur renvoyée.

Mais il existe également un autre moyen d'accéder aux docstring : vous pouvez utiliser la fonction help(). Apportez une petite modification à votre code : remplacez l'appel de la fonction d'impression par la ligne suivante : aide(my_fun)

Exécutez le code et voyez ce qui se passe. Quelles sont vos conclusions?

Comme vous pouvez le voir, la sortie est plus longue et plus descriptive :

Aide sur les fonctions my_fun dans le module __main__:

my_fun a) et b)
    La ligne de résumé va ici.

    Une description plus élaborée de la fonction.

    Paramètres:
    A : Int (description)
    B: int (description)

    Retourne:
    int : Description de la valeur renvoyée.

Essayez maintenant d'accéder aux docstrings de l'une des fonctions intégrées de Python (par exemple, *print()*). Importez ensuite un module et accédez à la documentation du module. Expérimentez la méthode __doc__ et la fonction help(). Voyez quelles sorties vous obtenez et en quoi elles diffèrent les unes des autres. Utilisez-les pour en savoir plus sur les objets Python intégrés.