Bonnes pratiques logicielles

Introduction :

Ce cours porte sur les bonnes pratiques logicielles. Car le développement de programmes informatiques nécessite non seulement des connaissances techniques, mais également le respect de principes et de conventions pour la production et la maintenance logicielle.

Nous aborderons dans un premier temps la nécessité de produire un code conforme. Puis nous nous intéresserons aux différentes optimisations qu'il est possible d'apporter à un code existant. Enfin, nous soulignerons dans une troisième partie la nécessité de documenter et de maintenir le code.

1 Conformité

Astuce

L'écriture de textes repose sur des règles d'orthographe, de grammaire et de ponctuation. De la même manière, les langages informatiques ont une orthographe, une grammaire et obéissent à des règles de syntaxe.

Nous allons présenter dans cette première partie les principes et outils qui contribuent à la production d'un code conforme aux usages.

a. Nommage

La manière de **nommer les éléments du code** a une incidence directe sur sa lisibilité. Nous pouvons aisément l'illustrer de manière concrète avec les portions de code ci-après.

```
for x in y:
```

Ce code ci-dessus est bien moins explicite que le suivant, strictement équivalent pour l'ordinateur :

```
for lettre in chaine:
```

De même le code `m = sum(n) / len(n)` ne permet pas immédiatement de comprendre la nature de ce qui est calculé et stocké dans la variable. Le code suivant, strictement équivalent pour l'ordinateur, est beaucoup plus compréhensible pour un humain : `moyenne = sum(notes) / len(notes)`

L'intérêt de choisir des noms explicites s'applique également aux fonctions.

 Exemple

```
def cf(dc):
    return dc * 9 / 5 + 32
```

Dans cet exemple, on comprend que la fonction effectue un calcul ; mais, son nom n'étant pas explicite, il est difficile d'en deviner la nature. La variable manipulée par la fonction étant elle aussi nommée de manière sibylline, toute personne autre que l'auteur·e aura du mal à comprendre d'emblée à quoi sert cette fonction.

 Attention

> Pire, l'auteur·e lui·elle-même aura peut-être des difficultés à se relire s'il·elle doit se plonger dans son propre code après plusieurs semaines ou plusieurs mois.

En réalité ce code est celui d'une fonction utilisée dans le cours précédent sur la modularité.

```
def celsius_en_fahrenheit(degres_celsius):
    """Calcule la température en degrés Fahrenheit
    équivalente à la température en degrés Celsius fournie en argument
    selon la formule température en Fahrenheit = température en Celsius x 9/5 + 32.
    """
    return degres_celsius * 9 / 5 + 32
```

La **docstring** (ou **chaîne de documentation**) nous éclaire précisément sur le traitement effectué par la fonction. Mais, même en son absence, le code reste très compréhensible grâce aux nommages explicites de la fonction et de la variable qu'elle traite.

```
def celsius_en_fahrenheit(degres_celsius):
    return degres_celsius * 9 / 5 + 32
```

b. Règles stylistiques en Python

La production de code Python obéit à un certain nombre de règles. Elles servent de bases communes pour que l'ensemble des développeur·se·s uniformise la rédaction dans ce langage.

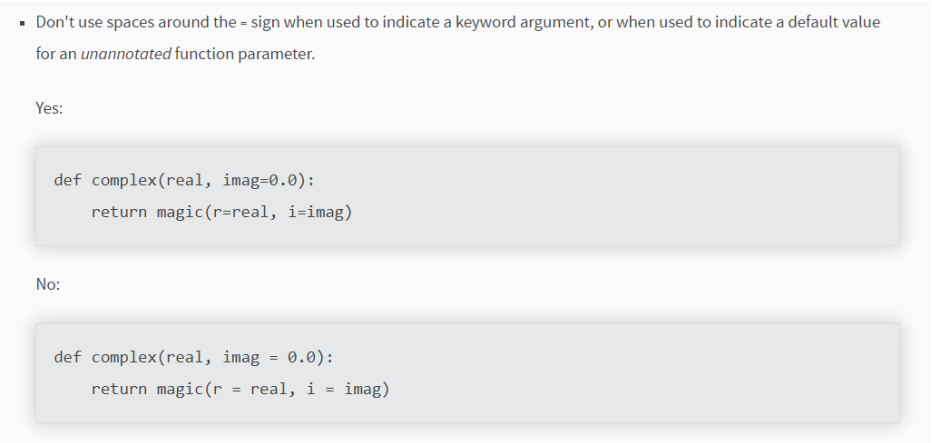
Ces règles sont définies dans un document appelé PEP 8 (PEP signifiant *Python Enhancement Proposal*, soit « proposition d'amélioration de Python »). Le document PEP 8 est un guide stylistique écrit en 2001 par plusieurs auteurs dont Guido van Rossum, l'inventeur du langage Python. Ce document précise les conventions en vigueur en matière de style pour ce langage.

Astuce

> Le PEP 8 est consultable (en anglais) dans son intégralité à l'adresse suivante : https://www.python.org/dev/peps/pep-0008/

Exemple

> - More than one space around an assignment (or other) operator to align it with another.
>
> Yes:
>
> ```
> x = 1
> y = 2
> long_variable = 3
> ```
>
> No:
>
> ```
> x             = 1
> y             = 2
> long_variable = 3
> ```



- Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter.

Yes:

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

No:

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

Le PEP 8 indique par exemple qu'on doit trouver une espace avant et après le signe `=` sauf pour les paramètres nommés des fonctions.

`moyenne = 12.7`

`def genere_mot_de_passe(longueur=12, majuscules=1, caracteres_speciaux=2):`

À retenir

Le respect des préconisations stylistiques du PEP 8 permet de produire un code dont la syntaxe est conforme aux usages pour le développement en Python.

Le guide PEP 8 précise entre autres :

- les conventions de **nommage** ;
- les modalités d'**indentation** ;
- la **longueur des lignes** ;
- l'**espacement** à appliquer entre les lignes.

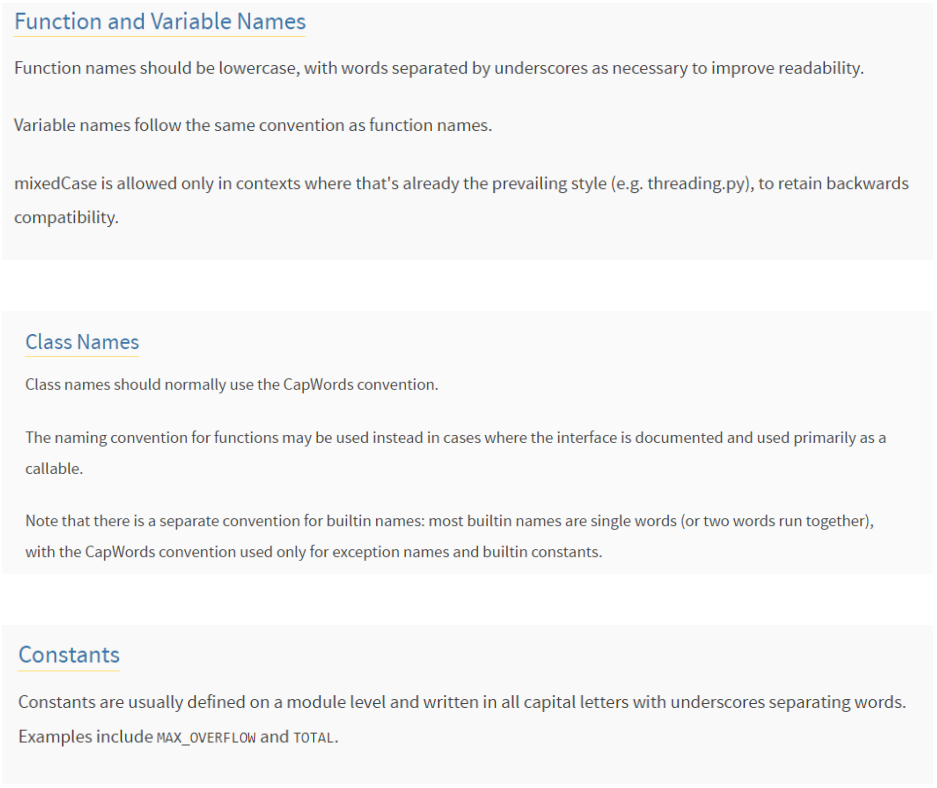
→ Ces préconisations permettent d'éviter que chaque développeur·se improvise sans concertation.

L'application de ces règles produit un code facilement lisible et compréhensible pour les autres développeur·se·s, habitué·e·s eux·elles aussi à appliquer ces mêmes règles. C'est une base qui sert de référence commune sur la manière d'écrire du code en Python.

Exemple

Voici quelques exemples de nommage en PEP 8 :

- le nom des variables, des fonctions et des méthodes s'écrit toujours en snake_case (mots en minuscules séparés par des underscore) ;
- le nom des classes s'écrit en toujours CamelCase (mots accolés avec la première lettre de chaque mot en lettre capitale) ;
- le nom des constantes s'écrit toujours en lettres capitales.



Function and Variable Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

Class Names

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

Mais tout n'est pas nécessairement défini de manière stricte pour autant. Ainsi le PEP 8 n'impose aucun style de guillemet particulier pour les chaînes de caractères : l'utilisateur·trice est invité·e à librement choisir des guillemets simples ou doubles, **et à s'y tenir** (sauf dans les cas où cela évite d'échapper des caractères).
En revanche pour les chaînes de caractères multi-lignes encadrées par des triples guillemets, PEP 8 précise qu'il faut toujours utiliser les guillemets doubles.

c. Outils d'aide à la mise en conformité syntaxique

Il existe des outils spécifiques pour aider le développeur à produire un code conforme.

Définition

> **Linter :**
>
> Un linter est un analyseur de code qui vérifie la conformité de sa syntaxe.

Définition

> **Linting :**
>
> Le linting est l'analyse d'un code effectuée par un linter.

Le périmètre d'analyse varie d'un linter à l'autre ; mais, d'une manière générale, ils signalent principalement les erreurs de syntaxe et le non-respect des règles stylistiques.

Un linter Python peut notamment :

- repérer les espaces excédentaires ou manquantes ;
- signaler les incohérences entre espaces et tabulations pour l'indentation ;
- indiquer si le nombre de lignes blanches requis (deux après les imports, une seule après une définition de fonction) n'est pas respecté ;
- alerter sur l'absence de chaîne de documentation.

Exemple

> Flake8 et Pylint sont deux linters Python fréquemment utilisés.

Les linters peuvent être utilisés en ligne de commande ou de manière intégrée à l'IDE (*Integrated Development Environnement*, soit « environnement de développement intégré ») du·de la développeur·se.

À retenir

> Lorsque le linter est intégré à l'éditeur de code, les erreurs ou problèmes sont directement balisés en regard du texte, avec des indications pour y remédier.

Le recours à un linter aide le·la développeur·se à adopter des bonnes pratiques et lui apporte la garantie que le code qu'il·elle produit est conforme aux règles syntaxiques et stylistiques du langage, contribuant ainsi à la qualité générale de son code.

Toutefois la qualité d'un code ne se résume pas au seul respect de la syntaxe. Après une première écriture aboutissant à la production du résultat recherché, un code doit souvent être remanié pour différentes raisons, ainsi que nous allons l'étudier maintenant.

2 Optimisation du code

La conformité stylistique du code est d'autant plus importante que le code produit est amené à être lu pour être compris et éventuellement modifié, par l'auteur·e ou par des tiers.

a. Réusinage

La reformulation de code est souvent désignée par le terme **réusinage** ou son équivalent anglais ***refactoring***.

Le code peut être modifié pour différentes raisons.

- Un code non conforme aux règles de style en vigueur doit être modifié afin de devenir conforme.
- Un code non conforme aux spécifications doit être adapté pour y répondre correctement.
- De nouveaux besoins peuvent avoir fait leur apparition, apportant des modifications ou des ajouts aux spécifications initiales.

- Un code peut se prêter à des optimisations, avec le recours à des spécificités du langage ou des choix d'implémentations qui rendent le programme plus modulaire, plus rapide ou moins consommateur de ressources.
- L'évolution de l'environnement dans lequel le code fonctionne peut nécessiter des adaptations pour des raisons de compatibilité, de performance ou de sécurité.

Chaque langage informatique a ses tournures et ses spécificités.

 Astuce

> Même si dans de nombreux cas on peut transposer assez littéralement des pratiques acquises avec un autre langage, il est préférable de respecter les usages en vigueur et d'employer les outils de référence de chaque langage, d'autant plus que ces outils ont en général été optimisés.

Dans le cas de Python la formule consacrée pour indiquer la conformité du code aux principes du langage consiste à le qualifier de « **pythonesque** ».

 Exemple

> Le parcours d'une liste peut s'effectuer par référence indicielle à ses éléments individuels :
>
> ```
> for i in range(len(liste)):
>     print(liste[i])
> ```
>
> De nombreux langages informatiques obligent à recourir à la notation indicielle, mais ce n'est pas le cas de Python. Ce code produit bien le résultat escompté mais il n'est pas du tout pythonesque. Un parcours de liste en Python s'écrit typiquement comme suit :
>
> ```
> for element in liste:
>     print(element)
> ```
>
> Le code est ainsi plus lisible. De même si on a besoin d'accéder simultanément à l'indice et à l'élément de la liste, le parcours s'effectue

de la manière suivante dans de nombreux langages :

```
for i in range(len(liste)):
    print(i, liste[i])
```

En Python on privilégiera plutôt la formulation suivante :

```
for i, element in enumerate(liste):
    print(i, element)
```

Les principes qui sous-tendent la programmation en Python ont été résumés dans un texte écrit en 1999 par Tim Peters, intitulé le « Zen de Python ». Ce texte de référence peut être affiché dans l'interpréteur Python avec la commande suivante :

```
import this
```

On obtient l'affichage du texte suivant :

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
```

```
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Astuce

> Une version française du Zen de Python est consultable sur la page Wikipedia correspondante : https://fr.wikipedia.org/wiki/Zen_de_Python.

À retenir

> Sans détailler chaque principe, on relèvera notamment la recommandation d'aérer le code et de favoriser sa lisibilité.

L'optimisation peut également porter sur la rapidité des traitements. Il existe souvent plusieurs manières d'aboutir à un même résultat, et dans le cas de traitements lourds ou portant sur de gros volumes de données, il peut s'avérer très bénéfique de **comparer les temps d'exécution de différentes approches**.

Dans le même ordre d'idée, la réduction de complexité algorithmique fait partie des motifs fréquents de réécriture de code. Illustrons-le avec un exemple.

Exemple

> Le code ci-après traite une liste de termes et vérifie, pour chacun d'entre eux, s'il figure parmi de nombreux termes d'une plus grande liste. Nous ferons l'analogie avec une meule de foin dans laquelle nous vérifions si plusieurs aiguilles sont ou non présentes.

```
compteur = 0

liste_aiguilles = [1,2,3]
meule_de_foin = [12,13,1,4,9,10,3,5,9,17,12,23,15,9,5,17]

for aiguille in liste_aiguilles:
        if aiguille in meule_de_foin:
            compteur = compteur +1
```

> `print(compteur)`
>
> À chaque tour de boucle l'aiguille provenant de la liste d'aiguilles appelée `liste_aiguilles` est recherchée dans la structure de données appelée `meule_de_foin`. Les listes étant très souvent utilisées comme conteneur de données par défaut, on considère que, dans ce code, `meule_de_foin` est une liste.

La complexité de cet algorithme est de l'ordre de $O(n^2)$, c'est-à-dire quadratique. Il imbrique en effet le test d'appartenance à la liste, de complexité $O(n)$, dans une boucle de complexité $O(n)$.

Ce code peut être grandement optimisé en optant pour un **set** à la place d'une liste (pour la meule de foin).

→ En effet, le test d'appartenance à un set étant de l'ordre de $O(1)$, cela ramène l'ensemble de ce code d'une complexité quadratique $O(n^2)$ à une complexité linéaire $O(n)$, avec un impact d'autant plus significatif sur les performances que les données traitées seront volumineuses.

```
compteur = 0

liste_aiguilles = [1,2,3]
meule_de_foin = set([12,13,1,4,9,10,3,5,9,17,12,23,15,9,5,17])

for aiguille in liste_aiguilles:
        if aiguille in meule_de_foin:
            compteur = compteur +1

print(compteur)
```

Notons que si l'on peut optimiser le code lui-même, on peut aussi parfois optimiser la nature des données qu'il doit traiter. Cela peut répondre, par exemple, à l'objectif de limiter l'espace mémoire occupé par le programme.

b. Couverture de code

Le recours à des tests est très utile pour s'assurer que le code produit bien les résultats attendus. Les tests unitaires ont fait l'objet d'un cours

spécifique en seconde.

Ces tests permettent à la fois de vérifier que le code produit bien le résultat attendu par rapport aux spécifications, mais aussi de s'assurer que le code continue de bien fonctionner après modification. Ceci pour éviter des régressions involontaires à la faveur d'un réusinage dont les conséquences n'auraient pas été bien mesurées.

Il apparaît donc important de disposer de jeux de tests permettant de le vérifier.

 Définition

> **Couverture de code :**
>
> La couverture de code est une mesure (exprimée en pourcentage) indiquant la proportion de code faisant l'objet de tests.
>
> → Le pourcentage de couverture de code traduit le taux de code couvert par les tests.

Les outils de couverture de code peuvent utiliser différentes méthodes de calcul pour évaluer cette couverture selon différents axes, mais l'essentiel est de retenir qu'il s'agit d'une mesure quantitative.

On comprend aisément que la seule présence de tests n'est pas une garantie de qualité, mais seulement de quantité. La couverture de code n'est donc pas un indicateur absolu : il ne permet pas de tirer de conclusion définitive sur la qualité et la pertinence des tests mis en place en regard du code.

c. Gestion des exigences et la certification CMMI

Un programme informatique est censé répondre à un certain nombre d'exigences, à l'instar d'objets plus concrets.

 Exemple

L'objet qui doit être conçu devra répondre à plusieurs exigences.

- **Exigence fonctionnelle** : griller des tartines de pain de dimensions maximales $20 \times 20\,\mathrm{cm}$
- **Exigence d'ergonomie** : être simple d'utilisation
- **Exigence d'apparence** : présenter un design épuré
- **Exigence de performance** : prendre au maximum $10\,\mathrm{s}$. pour griller un toast
- **Exigence de sécurité** : l'utilisateur ne doit pas risquer de se brûler ou de s'électrocuter en le manipulant.

Les exigences étant de natures diverses, il importe de définir les critères qui vont permettre de vérifier que l'application développée les respecte. C'est sur la base de ces exigences et de ces critères de validation qu'idéalement le jeu de tests doit être élaboré. Il est préférable de définir les tests en même temps que les exigences, sans attendre le développement. Cela permet en effet de vérifier que, d'une part, les demandes du client ont bien été comprises et que, d'autre part, elles sont testables.

À retenir

La démarche CMMi (*Capability Maturity Model Integrated*) est un exemple de modèle méthodologique pour la gestion de projet qui prend en compte spécifiquement la gestion des exigences.

d. Développement collaboratif

Certains développements logiciels sont effectués de manière individuelle. D'autres sont effectués de manière collaborative, notamment dans les entreprises de services numériques où plusieurs développeur·se·s travaillent simultanément sur un même projet. C'est également le cas pour un certain nombre de projets open-source qui sont maintenus par un noyau de contributeurs réguliers.

Le développement logiciel collaboratif nécessite que tous les participants s'accordent sur un certain nombre de conventions et s'organisent dans la répartition des tâches.

Exemple

> Des outils dédiés, tels Git, permettent la gestion de versions successives et la fusion de modifications (appelées « branches ») effectuées en parallèle par différentes personnes.

Abordons maintenant deux autres aspects importants du développement logiciel : sa documentation et sa maintenance.

3 Documentation et maintenance

a. Documentation

Parfois négligée, la **documentation du code** est pourtant un investissement rentable : elle facilite l'exploitation des fonctionnalités par les autres développeur·se·s, et par l'auteur·e lorsqu'il·elle doit se replonger dans son propre code pour l'optimiser ou le maintenir.

Les chaînes de documentation ou *doctstrings* constituent un moyen facile et rapide de documenter la production logicielle directement au niveau du code.

Astuce

> Au-delà d'une description factuelle des traitements opérés par le code, il ne faut pas hésiter à enrichir la documentation avec des exemples concrets qui correspondent à des cas d'usage simples et fréquents.

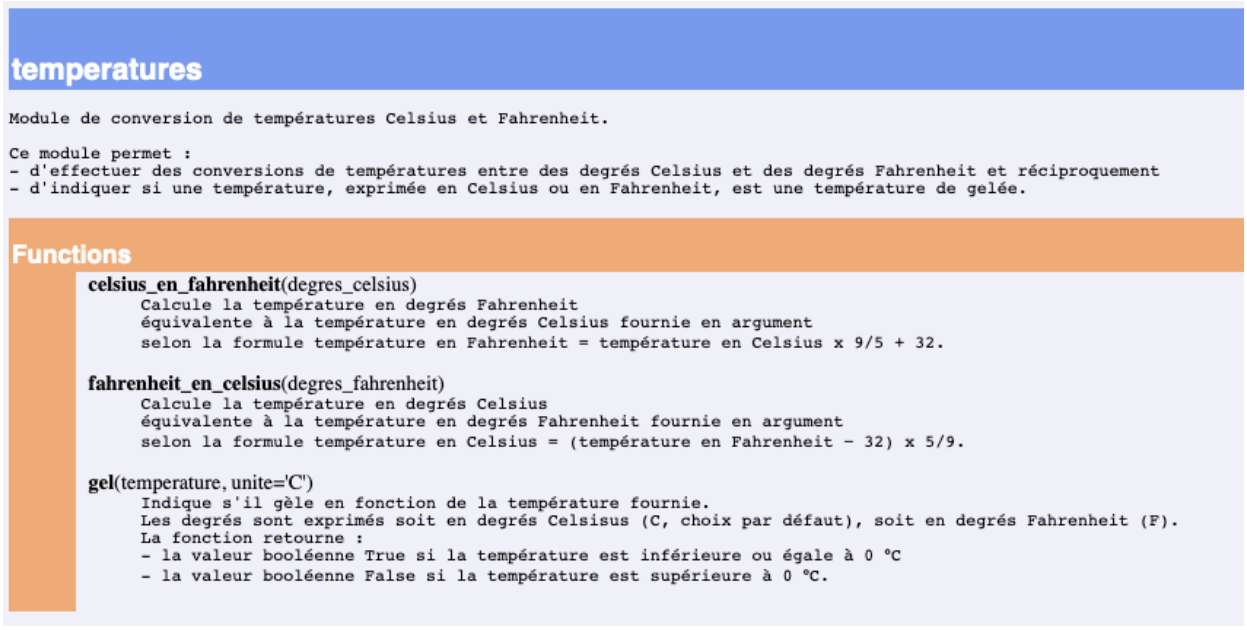
Cette documentation intégrée au code peut également être rendue accessible indépendamment du code à l'aide d'outils dédiés.

Exemple

> Pydoc est un module de documentation faisant partie de la bibliothèque standard. Pydoc fournit en ligne de commande un accès équivalent à la documentation obtenue avec l'instruction `help()` depuis un interpréteur Python.

Ce module permet non seulement d'accéder à l'ensemble de la documentation et d'effectuer des recherches sur le langage et ses bibliothèques, mais aussi de générer des fichiers de documentation au format texte ou HTML, et même un serveur HTTP local pour la consultation de la documentation.

Exemple

> 
>
> temperatures
>
> Module de conversion de températures Celsius et Fahrenheit.
>
> Ce module permet :
> - d'effectuer des conversions de températures entre des degrés Celsius et des degrés Fahrenheit et réciproquement
> - d'indiquer si une température, exprimée en Celsius ou en Fahrenheit, est une température de gelée.
>
> Functions
>
> **celsius_en_fahrenheit**(degres_celsius)
> Calcule la température en degrés Fahrenheit
> équivalente à la température en degrés Celsius fournie en argument
> selon la formule température en Fahrenheit = température en Celsius x 9/5 + 32.
>
> **fahrenheit_en_celsius**(degres_fahrenheit)
> Calcule la température en degrés Celsius
> équivalente à la température en degrés Fahrenheit fournie en argument
> selon la formule température en Celsius = (température en Fahrenheit - 32) x 5/9.
>
> **gel**(temperature, unite='C')
> Indique s'il gèle en fonction de la température fournie.
> Les degrés sont exprimés soit en degrés Celsisus (C, choix par défaut), soit en degrés Fahrenheit (F).
> La fonction retourne :
> - la valeur booléenne True si la température est inférieure ou égale à 0 °C
> - la valeur booléenne False si la température est supérieure à 0 °C.

> Cette page web a été générée par Pydoc directement à partir des différentes chaînes de documentation (*docstrings*) présentes dans le code du module « temperatures » développé dans le cours précédent sur la modularité.

Cette documentation au format HTML peut ensuite facilement être publiée sur le web ou sur un intranet pour proposer une ressource de documentation en ligne.

Astuce

> Pour des besoins de documentation plus élaborés, il existe un outil plus complet, écrit lui aussi en Python : le générateur de documentation **Sphinx**. Cet outil peut transformer des fichiers au format de balisage reStructuredText en différents formats parmi lesquels HTML, LaTeX, Epub, PDF, etc.

Enfin, la documentation en ligne d'une bibliothèque logicielle ne se limite pas nécessairement à l'aide intégrée ou à des documentations sur la description technique de ses composants.
Certain·e·s auteur·e·s proposent également des démonstrations et des tutoriels en ligne, sous forme textuelle ou vidéo, afin de faciliter la prise en main des outils qu'il·elle·s ont développés.

b. Maintenance

La vie d'un logiciel ne s'arrête pas à sa mise en production après validation de tous les tests. Il est nécessaire de maintenir la capacité de ce logiciel à fonctionner dans la durée.

Définition

> **Maintenance logicielle :**
>
> La maintenance logicielle désigne l'ensemble des actions nécessaires au bon fonctionnement d'un logiciel après sa mise en production.

La maintenance peut porter sur différents aspects susceptibles d'affecter le bon fonctionnement d'un logiciel, et notamment :

- des corrections de bugs découverts après la mise en production ;
- des améliorations ou modifications de fonctionnalités à la demande de l'utilisateur ;
- l'anticipation ou la prise en compte de l'évolution de la plateforme hébergeant le logiciel ou des composants requis par le faire fonctionner ;
- la migration sur une autre plateforme.

La mise à jour d'un système d'exploitation par son éditeur·rice ou d'une bibliothèque utilisée par le logiciel peuvent également entraîner des modifications du code, soit pour bénéficier d'améliorations de performance ou de sécurité, soit même pour pouvoir continuer à fonctionner en cas d'incompatibilité.

 Exemple

> Parmi les événements importants liés à la maintenance logicielle dans l'écosystème Python, on mentionnera la fin de vie de Python 2, survenue début 2020.

La version 2 de Python est apparue en 2000 et la version 3 en 2006. Les développeur·se·s du langage avaient annoncé en 2008 que la version 2 serait en fin de vie en 2015, afin de pouvoir se concentrer sur le développement de la version 3.

En 2015 un report a été décidé jusqu'au 1er janvier 2020 afin de laisser aux développeur·se·s le temps de faire évoluer leur code vers Python 3. Depuis cette date la version 2 de Python est officiellement obsolète et n'est plus supportée ni maintenue.

→ La fin de vie de la version 2 nécessite de porter le code Python 2 existant en Python 3 : on appelle cela le « portage de code ».

 À retenir

> Le portage de code est un terme technique qui désigne la transposition d'un code initial dans un autre environnement ou éventuellement un nouveau langage.

 Astuce

> la différence la plus visible entre du code écrit en version 2 ou en version 3 de Python se situe au niveau des appels à `print`, dépourvus de parenthèses en Python 2 alors qu'elles sont obligatoirement présentes en Python 3.

```
print "hello world" # Python 2
print("hello world") # Python 3
```

La documentation et la maintenance, parfois négligées, sont pourtant essentielles pour faciliter et entretenir le bon fonctionnement d'un logiciel dans la durée.

> Conclusion :
>
> Le développement logiciel repose sur un ensemble de bonnes pratiques visant à produire et à maintenir un code informatique conforme aux recommandations relatives au code utilisé, optimisé, documenté et répondant aux exigences du clients. Ces bonnes pratiques passent par le respect de règles et de conventions et peuvent également être empruntées à des méthodes de gestion de projet. De plus, des outils spécialisés aident le·la développeur·se à dans sa démarche.