Mise au point logicielle

Introduction :

La mise au point logicielle repose sur la connaissance d'un certain nombre de principes et de bonnes pratiques de développement, mais aussi sur la prise en compte des spécificités du langage choisi pour l'implémentation. Aussi, dans ce cours, nous présenterons les bonnes pratiques logicielles en soulignant certaines spécificités du langage Python.

Nous aborderons dans un premier temps les effets de bord, puis nous nous intéresserons aux traitements conditionnels et enfin aux comparaisons, chaque fois sous l'angle de la mise au point logicielle.

1 Effets de bord

Rappel

Un langage à **typage dynamique** n'impose pas que le type des variables soit défini. Le type est alors déterminé à la volée au moment de l'interprétation du code.

a. Typage dynamique

Python étant un langage à typage dynamique, il n'impose pas de spécifier le type d'une donnée, et on peut à loisir changer le type de donnée d'une variable.

Exemple

```
a = 'hello world'
a = a.split() # a désigne la liste ['hello', 'world']
a = 12
a = True
```

Attention

De tels changements seraient en revanche **impossibles dans les langages à typage statique** qui imposent de définir précisément le type de données d'une variable et de le respecter ensuite.

Le typage dynamique nous permet d'envisager un même traitement applicable à plusieurs types de données.
Illustrons-le avec une procédure simple qui imprime chacun des éléments d'un itérable.

```
def enumere(iterable):
    for element in iterable:
        print(element)
```

Testons ensuite notre procédure avec plusieurs types d'itérables.

```
enumere('1497') # chaine de caractères
```

On obtient l'affichage suivant :

```
1
4
9
7
```

On peut obtenir le même affichage d'autres types de données.

```
enumere([1, 4, 9, 7]) # liste
enumere((1, 4, 9, 7)) # tuple
enumere({1, 4, 9, 7}) # set
enumere({'1': 5, '4': 0, '9': 8, '7': 2}) # dictionnaire
```

Ces quatre appels de procédure produisent tous l'affichage suivant :

```
1
4
9
7
```

Mais dans certains cas un traitement différencié peut s'avérer nécessaire en fonction de la nature des données traitées. Dans l'exemple qui précède, nous avons traité indifféremment des types mutables (liste, set, dictionnaire) et des types non mutables (chaîne de caractères, tuple).

→ Les types mutables autorisent l'affectation ou la suppression d'éléments mais ce n'est pas le cas des types non mutables qui génèrent une erreur si on tente une affectation ou une suppression d'élément.

```
t = ('a', 'b', 'c') # tuple
print(t[1])
t[1] = 'z'
# affiche TypeError: 'tuple' object does not support item assignment

del(t[2])
# affiche TypeError: 'tuple' object does not support item deletion
```

Si nous voulions apporter des modifications aux itérables, il faudrait prévoir des traitements adaptés en fonction de leur mutabilité. De même certains d'entre eux sont ordonnés (chaîne de caractères, liste, tuple) et d'autres ne le sont pas (dictionnaire, set). La souplesse apportée par le typage dynamique ne dispense donc pas d'adapter les traitements aux types de données possiblement traités.

À retenir

> Le choix du type de structure de donnée doit s'effectuer en fonction des besoins et des caractéristiques désirées, notamment d'ordre et de mutabilité.

b. Mutabilité

Certaines structures de données ont été conçues avec la caractéristique d'être **mutables**, c'est-à-dire modifiables.

Il est nécessaire d'être vigilant·e sur cette caractéristique en programmation fonctionnelle où une fonction ne doit pas créer d'**effet de bord**, c'est-à-dire qu'elle ne doit pas modifier d'éléments extérieurs à son environnement local.

Voici un exemple de mutation appliquée à un dictionnaire par une fonction.

 Exemple

> Nous disposons d'un dictionnaire composé de clés associées à des valeurs numériques entières, positives ou négatives. Nous souhaitons créer une fonction qui reçoive en entrée un tel dictionnaire et retourne en sortie un dictionnaire où les valeurs négatives ont été remplacées par zéro.
>
> ```
> def plancher(dictionnaire):
>     for cle, valeur in dictionnaire.items():
>         if valeur < 0:
>             dictionnaire[cle] = 0
>     return dictionnaire
>
> entrees = {'A': 12, 'B': 21, 'C': -2, 'D':37, 'E':-99, 'F': 6}
>
> print(plancher(entrees))
> # affiche {'A': 12, 'B': 21, 'C': 0, 'D': 37, 'E': 0, 'F': 6}
>
> print(entrees)
> # affiche {'A': 12, 'B': 21, 'C': 0, 'D': 37, 'E': 0, 'F': 6}
> ```
>
> On constate que la fonction `plancher()` a bien retourné le résultat attendu, mais elle a aussi modifié le dictionnaire qui a servi de base au traitement.
>
> Pour éviter cette mutation notre fonction doit travailler sur une copie et pas sur l'original.
>
> ```
> def plancher(dictionnaire):
>     copie = dictionnaire.copy()
>     for cle, valeur in copie.items():
>         if valeur < 0:
>             copie[cle] = 0
> ```

```
        return copie
entrees = {'A': 12, 'B': 21, 'C': -2, 'D':37, 'E':-99, 'F': 6}

print(plancher(entrees))
# affiche {'A': 12, 'B': 21, 'C': 0, 'D': 37, 'E': 0, 'F': 6}

print(entrees)
# affiche {'A': 12, 'B': 21, 'C': -2, 'D': 37, 'E': -99, 'F': 6}
```

→ La version réécrite de notre fonction `plancher()` a bien effectué le traitement demandé, mais elle n'a pas modifié le dictionnaire des entrées qui a servi de base au traitement.

⚠ Attention

La copie de structures de données mutables avec la méthode `copy()` consiste en une copie superficielle des données (« *shallow copy* » en anglais). Si ces structures de données mutables contiennent elles-mêmes des données mutables, telles que des listes ou des dictionnaires, il est nécessaire d'effectuer ce qu'on appelle une copie profonde (« *deep copy* » en anglais).

La bibliothèque `copy` faisant partie de la bibliothèque standard de Python, elle propose les méthodes correspondantes.

c. Débordements

La notion de **débordement** est associée à la notion de **pile**.

📖 Définition

Pile :

Une pile est un conteneur dans lequel il est possible d'ajouter (empiler) et de retirer (dépiler) des éléments.

La logique de fonctionnement d'une pile est résumée par le sigle LIFO (pour « *Last In, First Out* », c'est-à-dire « dernier entré, premier sorti »).

 À retenir

> Avec une pile **on ne peut retirer que le dernier élément empilé**. Si on veut accéder à des éléments situés en-dessous, il faut d'abord retirer (dépiler) un par un ceux qui se trouvent au-dessus.

 Astuce

> On peut faire l'analogie avec une pile de cartons lourds qui ne pourraient être manipulés qu'individuellement.

Des débordements peuvent survenir dans deux cas :

- **débordement positif** quand la pile, limitée en taille, est pleine et qu'on tente d'ajouter un nouvel élément ;
- **débordement négatif** quand la pile est vide et qu'on tente de retirer un élément.

En Python on peut assez facilement modéliser une pile en s'appuyant sur une liste. Si certains langages imposent de définir à l'avance une taille de tableau, et donc un nombre fixe d'éléments, ce n'est pas le cas de Python dont les listes peuvent par défaut être étendues à loisir. Mais on pourrait facilement implémenter une telle restriction sur une liste pour disposer d'une pile à capacité limitée.

On peut constater un débordement négatif sur une liste si on appelle la méthode `pop()` sans nous assurer au préalable que la liste contient au moins un élément.

```
nombres = [12, 7, 9]
```

```
print(nombres.pop())
# affiche 9
```

```
print(nombres.pop())
# affiche 7

print(nombres.pop())
# affiche 12

print(nombres.pop())
# affiche IndexError: pop from empty list
```

Un test préalable est donc nécessaire pour s'assurer que la liste n'est pas complètement vide avant de tenter de la dépiler.

Astuce

> On peut implémenter ce test en exploitant la différence de valeur booléenne entre une liste vide et une liste qui ne l'est pas.

```
print(bool([]))
# affiche False
print(bool([1]))
# affiche True
```

Cette différence nous permet de produire facilement un code qui dépile complètement une liste sans erreur à la fin.

```
nombres = [12, 7, 9]
while nombres:
    print(nombres.pop())
# absence d'erreur, on sort de la boucle quand la pile est vide
```

À retenir

> Les effets de bord peuvent perturber la mise au point logicielle. Il est important de les connaître afin d'éviter de les subir de manière involontaire.

Les effets de bord ne sont pas les seules sources d'erreurs logicielles. Les instructions conditionnelles doivent elles aussi être employées de manière adéquate.

2 | Instructions conditionnelles

Les instructions conditionnelles permettent à nos algorithmes d'effectuer des traitements distincts en fonction de conditions particulières. Nous évaluons si certaines conditions sont remplies et si c'est le cas nous effectuons le traitement correspondant.

Rappel

> Ces instructions conditionnelles sont articulées en Python autour d'au moins un `if` (« si »), optionnellement associé à un ou plusieurs `elif` (signifiant « sinon si ») et optionnellement à un `else` (« sinon »).

Nous allons présenter plusieurs cas de figures d'**erreurs fréquemment commises** avec des instructions conditionnelles.

a. Séquence incomplète

Voyons tout d'abord une séquence incomplète.

```
x = 5

if x < 0:
        y = - 20
elif x < 10:
        y = 21
elif 10 < x < 12:
        y = 35
elif x >= 12:
        y = 37

print(y)
# affiche 21
```

Mais si nous relançons ce même code en affectant la valeur 10 à `x`, nous obtenons une erreur.

```
NameError: name 'y' is not defined
```

Cet ensemble d'instructions conditionnelles ne couvre pas tous les cas de figure. En effet le cas où `x` est égal à 10 n'est traité par aucune des branches : aucune valeur n'est affectée à `y`.

Une clause `else` permet de fournir des instructions quand aucune autre condition n'a été remplie, mais dans le cas présent où toutes les valeurs de `x` autres que 10 ont été traitées, l'erreur viendrait plutôt d'un comparateur `<=` qui aurait été saisi en `<` dans le premier ou le deuxième `elif`.

b. Séquence complète avec ordre incorrect des conditions

On peut avoir envisagé tous les cas de figure mais avoir commis une erreur dans l'ordre de traitement des conditions.

On cherche à implémenter un algorithme qui affecte une valeur y en fonction d'une valeur x selon le tableau suivant :

Valeurs de x	Valeurs de y
$x <= 20$	$y = -20$
$x > 20$	$y = 21$
$x > 30$	$y = 35$
$x > 50$	$y = 37$

La transposition littérale du tableau par ordre chronologique nous fournit le code suivant :

```
if x <= 20:
    y = - 20
elif x > 20:
    y = 21
elif x > 30:
    y = 35
elif x > 50:
```

```
    y = 37
```

Ce code fonctionnera uniquement pour certaines valeurs car dès que `x` dépasse 20, la condition du premier `elif` devient vraie, et les branches suivantes ne seront jamais traitées. Il faudrait donc soit réordonner les inégalités existantes, soit créer des conditions sous forme d'encadrement, du type : `elif 20 < x <=30`.

La façon dont le problème est défini n'est pas toujours directement transposable en algorithme de manière littérale.

c. Traitement conditionnel d'égalités

Le problème d'ordre décrit précédemment avec des inégalités disparaît avec des égalités, puisque par définition une variable ne peut pas simultanément être égale à plusieurs valeurs. Toutefois, on peut être confronté·e à des problématiques proches avec l'évaluation simultanée d'égalités portant sur plusieurs valeurs.

Exemple

> Énoncé du problème :
>
> - si $x = 2$ ou $y = 5$, alors $z = 35$
> - si $x = 2$ et $y = 5$, alors $z = 15$
> - sinon $z = 20$

Avec un tel énoncé, qui s'apparente à du pseudo-code, la transposition paraît assez immédiate.

```
if x == 2 or y == 5:
    z = 35
elif x == 2 and y == 5:
    z = 15
else:
    z = 20
```

Pourtant ce code ne fonctionnera pas comme prévu.

Il est certain qu'une valeur sera affectée à `z` grâce à la clause `else` qui permet de traiter le cas où aucune des évaluations d'expressions antérieures n'est vraie.

On comprend par ailleurs à la lecture de ce code l'intention recherchée d'affecter la valeur 15 à `z` quand `x` vaut 2 et `y` vaut 5.

Attention

> Mais l'embranchement correspondant, celui du premier `elif`, n'est jamais atteint.

En effet, dans ce cas de figure, la condition évaluée dans le `if` est vraie : il suffit qu'une seule des deux variables soit égale à la valeur testée, mais avec le « ou » logique la condition reste vraie aussi quand les deux expressions sont vraies. Comme cette condition est évaluée avant la condition comportant le « et » logique, cette dernière n'est pas évaluée.

Pour obtenir le résultat souhaité, on doit donc **inverser l'ordre des deux évaluations** : on commence par vérifier si les deux égalités sont vraies simultanément, et si ce n'est pas le cas, on évalue ensuite si au moins une des deux l'est.

```
if x == 2 and y == 5:
    z = 5
elif x == 2 or y == 5:
    z = 35
else:
    z = 20
```

d. Sécurité des traitements

Lorsqu'un algorithme doit déterminer selon un ensemble de conditions si une permission est accordée ou non, il est préférable de toujours considérer comme point de départ le niveau minimum de permission, plutôt que de faire le contraire. Ce niveau minimum pourra éventuellement être augmenté si les conditions sont réunies.

Ainsi dans le cas où on cherche à déterminer si un mot de passe est, ou non, conforme à un certain nombre de critères, **on commencera par considérer qu'il n'est pas conforme**. On évaluera ensuite individuellement chaque condition (par exemple longueur du mot de passe, présence conjointe de majuscules et de minuscules, présence de caractères spéciaux etc.), et on ne validera le mot de passe que si toutes les conditions sont bien réunies.

De la même manière lorsqu'on implémente une fonctionnalité d'ajout d'utilisateur·rice·s pour l'administrateur·rice d'une plateforme, et que ces visiteur·se·s peuvent avoir plusieurs rôles correspondant à des niveaux de permissions croissants (par exemple :visiteur·se, membre inscrit·e, modérateur·rice, administrateur·rice), on fera en sorte que le rôle par défaut d'un·e nouvel·le utilisateur·rice soit « visiteur » plutôt qu'« administrateur ». De cette manière si l'administrateur·rice oublie de changer les droits par défaut pour en faire un usager plus privilégié, celui-ci obtiendra seulement les droits d'un visiteur : ses droits seront trop restreints, ce qui est assurément moins problématique en termes de sécurité que des droits trop étendus.

À retenir

> Les traitements conditionnels nécessitent une vigilance particulière afin de s'assurer que tous les cas de figure possibles sont traités, et que les traitements prévus pour chaque cas spécifique sont bien effectués comme prévu.

Nous allons maintenant nous intéresser aux problèmes pouvant survenir en effectuant des comparaisons.

3 | Comparaisons

De nombreux algorithmes nécessitent de comparer des valeurs entre elles. Nous avons montré précédemment qu'il était important de prévoir tous les cas de figure de manière exhaustive et sans commettre d'erreur dans l'ordre des évaluations pour les inégalités. Si l'ordre n'a pas d'importance pour les égalités simples, la notion même d'égalité doit être nuancée.

a. Égalité des nombres réels

La comparaison d'entiers (type `int`) ne pose pas de problèmes particuliers.

```
print(1 + 1 == 2)
# affiche True
```

```
print(1 + 1 + 1 == 3)
# affiche True
```

Il convient en revanche d'être prudent avec les nombres flottants (type `float`).

```
print(0.1 + 0.1 == 0.2)
# affiche True
```

```
print(0.1 + 0.1 + 0.1 == 0.3)
# affiche False
```

Ce résultat surprenant et inattendu tient à la manière dont les ordinateurs gèrent les nombres flottants. Sans entrer dans les détails techniques, il faut retenir que certaines valeurs sont seulement des valeurs **approchées**. Nous pouvons l'illustrer en demandant à Python d'afficher la valeur du nombre réel 0.3 avec un très grand nombre de décimales.

```
print('{:.80f}'.format(0.3))
# affiche
0.10000000000000000055511151231257827021181583404541015625000000000000000000000000000000
```

À retenir

> Les tests d'égalité stricte sont donc à éviter lorsqu'on manipule des nombres réels. On passera par des arrondis ou par des comparaisons prenant en compte un faible écart que l'on aura choisi de tolérer.

```
print(round(0.1 + 0.1 + 0.1, 2) == round(0.3, 2))
# affiche True
```

```
ecart = 0.1 + 0.1 + 0.1 - 0.3
ecart_tolere = 0.01
```

```
print('{:.80f}'.format(ecart))
# affiche
0.00000000000000005551115123125782702118158340454101562500000000000000000000000000
print(abs(ecart) < ecart_tolere)
# affiche True
```

Astuce

> La bibliothèque de tests unitaires `unittest`, incluse dans la bibliothèque standard, comporte une méthode d'assertion de quasi-égalité appelée `AssertAlmostEqual` reposant sur ce principe.
> Cette méthode permet de préciser un ordre de grandeur de l'écart (appelé conventionnellement delta) acceptable pour la comparaison.

b. Identité et équivalence

Lorsqu'on compare deux variables, il est important de comprendre sur quoi porte la comparaison.

On doit distinguer deux notions :

- **l'égalité structurelle**, qui indique si le contenu des objets est identique ;
- **l'égalité physique**, qui indique s'il s'agit ou non du même objet.

Illustrons ces deux notions avec un exemple.

Exemple

```
liste1 = ['a', 'b', 'c']
liste2 = liste1
liste3 = ['a', 'b', 'c']

print(liste1 == liste2)
# affiche True

print(liste1 == liste3)
# affiche True
```

Les variables `liste1`, `liste2` et `liste3` ont bien le même contenu.

```
print(liste1 is liste2)
#affiche True

print(liste1 is liste3)
# affiche False
```

Les variables `liste1` et `liste2` font référence à un même objet commun. En revanche `liste3` est un objet distinct. On peut le matérialiser en consultant l'adresse mémoire pointée par les trois variables. On constate que `liste1` et `liste2` désignent bien la même adresse, contrairement à `liste3` .

```
print(liste1.__repr__)
# affiche repr' of list object at 0x11191fb08>
print(liste2.__repr__)
# affiche repr' of list object at 0x11191fb08>
print(liste3.__repr__)
# affiche repr' of list object at 0x112339608>
```

Toute modification effectuée sur `liste1` affectera obligatoirement `liste2` puisqu'il s'agit d'un seul et même objet. En revanche `liste3` étant un objet distinct, il pourra être modifié de manière indépendante.

En fonction de l'objectif recherché, on vérifiera soit l'identité soit l'équivalence entre deux objets comparés.

La connaissance des sources d'erreur évoquées précédemment permet au·à la programmeur·se d'anticiper ses erreurs. En ayant en tête ces sources d'erreur au moment de l'écriture du programme, il·elle gagnera un temps précieux dans la mise au point de son programme. Cependant, on n'échappera pas forcément à la manifestation d'un bug. Il est alors nécessaire de procéder méthodiquement pour corriger celui-ci. Regardons cela de plus près.

4 | Démarche de correction d'un bug

a. Identifier le bug

Lorsqu'un bug apparaît, il importe, dans un premier temps, de procéder à son identification. Pour cela, il est nécessaire de procéder en trois étapes :

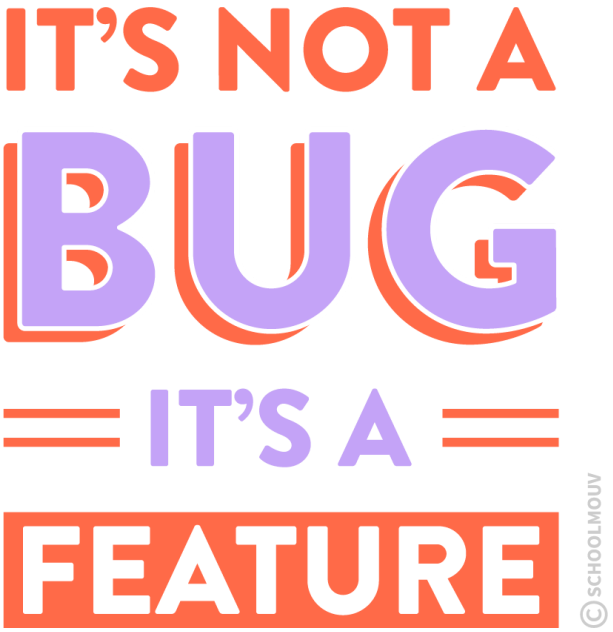
1. être en mesure de reproduire le bug, donc d'identifier les circonstances dans lesquelles celui-ci apparaît ;
2. bien cerner le résultat normalement attendu, ce qui impose en particulier de reprendre connaissance du détail des exigences relatives au programme. Le but étant de corriger le programme dans le strict respect de celles-ci ;
3. les étapes 1 et 2 étant passées, il est alors temps de localiser le bug, c'est-à-dire la partie de code fautive, sans rentrer au niveau de la ligne de code à ce stade-là.

On commence à identifier le module qui génère le bug. Le bug peut être dû à une erreur de programmation dans le module lui-même, ou au fait qu'il travaille avec des données en entrée qui sont elles-mêmes erronées. Une fois le module fautif identifié, on resserre le champ de recherche en se focalisant sur ce module et en cherchant à identifier le sous-module, puis la fonction ou la portion de code fautive.

b. Corriger le bug

Une fois le bug identifié on procède en quatre étapes :

1. corriger le bug ;
2. enrichir le jeu de tests du programme pour couvrir le cas qui a généré le bug ;
3. profiter de son intervention dans le code pour l'améliorer, le rendre plus clair ;
4. tester la correction du bug et vérifier également que l'on n'a pas commis de régression, c'est-à-dire engendré d'autres bugs.



Conclusion :

La mise au point logicielle nécessite de respecter un certain nombre de bonnes pratiques de développement et aussi de connaître les spécificités du langage utilisé pour effectuer ce développement.

Nous l'avons illustré avec le langage Python en abordant successivement les effets de bords, les traitements conditionnels et les comparaisons. Nous avons pu constater que l'écriture d'algorithmes corrects reposait sur des choix éclairés de structures de données prenant en compte leurs différentes caractéristiques, mais aussi sur notre capacité à implémenter une problématique qui n'est pas toujours littéralement transposable, même quand elle est exprimée de manière proche du code. Cependant, si le développeur est malgré tout confronté à un bug, il saura le corriger efficacement en suivant une démarche rigoureuse qui assurera capitalisation et non régression.
Pour terminer ce cours avec un peu d'humour, voici (ci-contre) une citation bien connue des développeurs.