Architecture (logicielle) 16 – Gestion des bugs

I – Types d'erreurs

Il existe **trois grandes catégories** d'erreurs.

→ **les erreurs de syntaxe** :

→ l'interpréteur ou le compilateur ne reconnaît pas l'instruction sur l'une des lignes car elle est mal formulé pour lui

→ Niveau de difficulté de détection : Facile

→ Correction : il suffit de lire ce que dit l'interpréteur ou le compilateur

→ Exemple : _____

```
def addition(a,b)
    return a + b
```

→ **les erreurs d'exécution** :

→ le code est correct mais c'est le contenu des variables ou des fichiers qui posera problème au moment où on les utilise.

→ le problème ne réside pas sur une ligne mais une interaction entre des contenus

→ Niveau de difficulté de détection : Difficile

→ Correction : soit on trouve le problème rapidement, soit il faut faire tourner le programme en mode pas à pas à l'aide d'un debuggeur.

→ Exemple : _____

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

addition(5, '10')
```

→ **les erreurs de logique (ou erreurs sémantiques)** :

→ le code est correct et ne provoque pas d'erreurs ! Néanmoins, il ne répond pas correctement.

→ le problème réside dans l'algorithme lui-même.

→ Niveau de difficulté de détection: Très Difficile

→ Correction : on doit faire des tests et tenter de trouver d'où vient le problème. La bonne méthode : avoir réaliser de petites fonctions et beaucoup de tests automatiques pour parvenir à localiser rapidement le problème. Et ensuite, chercher...

→ Exemple : _____

```
def multiplication(a,b):
    m = 0
    for x in range(b):
        m = m + a
        return m
```

II – Quelques erreurs typiques

→ **Expression booléenne mal exprimée**

→ Cause : mauvaise compréhension de l'interprétation de l'expression booléenne.

→ Exemple : On veut dire **a et (b ou c)**

→ On oublie que le ET est prioritaire sur OU

→ On écrit **a and b or C**

→ Que comprend l'interpréteur Python ?

_____

→ Type de l'erreur : _____

→ **Effets de bord non désirés**

→ Cause : modification d'une zone-mémoire via des alias.

→ Exemple :

```
a = [4, 10]
b = a
b.append(12)
```

→ Que contient b ? a ?

_____

→ Type de l'erreur : _____

→ Attention aux listes chaînées mutables

L1 : N0 → N1 → N2

L2 : N2 → N3 → N4

→ Quelle est la liste totale pour L1 :

_____

→ Que devient L2 si on rajoute … N4 → N5 ?

_____

→ Que devient L1 du coup ?

_____

→ **Débordement dans les tableaux**

→ Cause : usage d'un indice non disponible

→ Exemple :

```
t = [5, 10, 15]
dernier = t[len(t)]
```

Problème :

→ Type de l'erreur : _____

→ **Instruction conditionnelle non exhaustive**

→ Cause 1 : seul un bloc est valable

→ Cause 2 : on omet de gérer un intervalle

→ Exemple des appréciations :

```
def renvoyer_appreciation(note) :
    app = 'passable'
    if note > 12 :
        app = 'AB'
    elif note > 14 :
        app = 'B'
```

Pourquoi **renvoyer_appreciation(16)** renvoie 'AB' et pas 'B' ?

→ **choix des inégalités**

→ **Cause : confusion entre > et <**

→ **Cause : confusion entre ≥ et >…**

→ Souvent lié à un while et un not

→ Exemple ci-dessous

→ Cause de l'erreur :

→ Type de l'erreur : _____

```
def somme_4_premiers(t):
    s = 0
    i = 0
    while i <= 4 :
        s = s + t[i]
        i = i + 1
    return s

print(somme_4_premiers([1,2,3,4,5,7,11]))
```

→ **Borne finale de la boucle for**

→ **Cause : on oublie qu'on n'atteint pas la valeur indiquée** : on s'arrête juste avant.

→ Exemple

→ Type de l'erreur : _____

```
def somme_4_premiers(t):
    s = 0
    for i in range(3):
        s = s + t[i]
    return s

print(somme_4_premiers([1,2,3,4,5,7,11]))
```

→ **Oubli de l'incrémentation du variant**

→ **Cause : on oublie de faire varier le variant**

→ conclusion : l'infini et au-delà !

→ Type de l'erreur : _____

```
def somme_4_premiers(t):
    s = 0
    i = 0
    while i < 4 :
        s = s + t[i]
    return s
```

→ mauvaise initialisation d'un variant

→ **on oublie l'initialisation ou on lui donne une mauvaise valeur de départ**

→ Type de l'erreur : _____

```
def somme_4_premiers(t):
    s = 0
    i = 10
    while i < 4 :
        s = s + t[i]
        i = i + 1
    return s

def somme_4_premiers(t):
    s = 0
    while i < 4 :
        s = s + t[i]
        i = i + 1
    return s
```

→ **comparaisons et calculs entre flottants**

→ **Cause : les floats ne sont pas encodés de façon exacte en machine le plus souvent.**

→ Ne **JAMAIS** faire de tests d'égalité entre deux floats !

```
>>> 0.3 == (0.1*3)
False
```

→ Attention aux calculs en boucles avec des floats : un écart initial entre vraie valeur et valeur stockée peut s'amplifier au fil du temps.

```
a = 0.1
b = 0.3
c = b - 3*a
for d in range(1000):
    c = c*2
print(c)
```

A la fin, c contient environ $-5.948.10^{284}$!

→ **problèmes liés au typage**

→ **Cause : les opérateurs agissent différemment en fonction des types.**

→ On agit parfois sur un type de données alors qu'il s'agit d'un autre contenu.

```
a = 5
b = '2'
c = a*b
```
→ c contient _____

Mais on trouve également :

→ **le mauvais nommage de variables :**

→ n'utilisez jamais l minuscule par exemple : il ressemble trop à un 1.

→ un nom trop long peut être mal orthographié.

→ l**a mauvaise indentation dans Python :**

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

    def multiplication(a,b):
        return a*b
```

→ Problème ici :

III –Documentation

La documentation permet d'expliquer comment utiliser la fonction.

On la positionne **sous le prototype Python** de la fonction. On y trouve

→ les types de paramètres

→ les préconditions sur les paramètres d'entrées; ce qu'on doit respecter normalement.

→ les postconditions sur la sortie; ce qu'on peut attendre si les préconditions sont bonnes.

→ les effets de bords éventuels

→ des exemples d'utilisation

IV – Assertions

<u>Rappel</u> : dans un module, on trouve deux types de fonctions :
→ les fonctions d'interface : l'utilisateur qui importe le module a le droit de les utiliser
→ les fonctions privées du module : elles ne sont pas utilisables directement par un utilisateur.

Les assertions servent souvent à surveiller le respect des préconditions et/ou des postconditions lors de la phase « prototype ». Ensuite, on peut les <u>débrayer</u>.
Thonny : Run → Run current script in terminal
puis **$ python -O nom_du_script.py**

Principe :
→ **Python évalue l'assertion** (expression vraie normalement).
→ **Si elle est True**, on continue sans rien faire d'autres.
→ **Si elle est False**, on lève une EXCEPTION. Par défaut, cela interrompt l'exécution.

On peut placer un grand nombre d'assertions lors de la conception.
Une fois le module terminé, il faudra alors en supprimer de façon à ne pas ralentir l'exécution de façon aberrante.

On peut laisser quelques assertions bien choisies dans les fonctions internes d'un module si on travaille en <u>programmation offensive</u> : on veut interrompre le fonctionnement le plus tôt possible, avant que la mauvaise donnée ne corrompe ou détruise d'autres parties du système.

Il faut veiller à l'enchaînement des fonctions pour savoir ce qu'il est vraiment nécessaire de tester : inutile de tester une chose qui a déjà été vérifié ailleurs.

```
def inverse(a):
    '''Renvoie l'inverse de a, non nul
    :: param a(int)   :: un entier non nul
    :: return (float) :: l'inverse de a
    :: exemple ..
    >>> inverse(10)
    0.1

    '''
    # Préconditions
    assert type(a) == int
    assert a != 0

    r = 1 / a

    # Postcondition
    assert type(r) == float

    return r
```

V – Tester

Tester les fonctions permet de vérifier qu'elle réponde bien ce qu'il faut. Deux utilisations :
→ **lors de la conception** (on sait si on a fini ou pas)
→ **lors de l'utilisation** (veille automatique lorsqu'on fait des modifications)

En Python, il existe de multiples façons de réaliser cela.

<u>1 – Via le module doctest</u>

Ce module permet d'utiliser la documentation pour réaliser des tests automatiques.

```
def multiplication(a,b):
    '''Renvoie a*b
    :: param a(int) :: un entier
    :: param b(int) :: un autre entier
    :: return (int) :: a*b, un entier
    :: exemple ..
    >>> multiplication(5,10)
    50

    '''
    return a*b

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

<u>2 – En créant votre propre fonction de test</u>

```
def multiplication(a,b):
    '''Renvoie a*b
    :: param a(int) :: un entier
    :: param b(int) :: un autre entier
    :: return (int) :: a*b, un entier
    :: exemple ..
    >>> multiplication(5,10)
    50

    '''
    return a*b

def test_multiplication():
    assert multiplication(10,5) == 50

if __name__ == '__main__':
    test_multiplication()
```

<u>Rappel de la réalisation d'un projet</u> :

→ On réflechit au déroulé du programme
→ On écrit les algorithmes
→ On écrit les prototypes des fonctions
→ On écrit la documentation des fonctions (dont les exemples d'utilisations)
→ On fournit un jeu de tests au concepteur.
→ On fournit les algorithmes au concepteur et il les implémente dans le langage voulu : tant que le jeu de tests ne répond pas positivement, c'est que le travail n'est pas fini.