Mise au point d'un programme

Table des matières

1) Débogger un programme

La phase de débogage d'un programme, qui consiste à rechercher les erreurs de programmation, est très gourmande en temps. ceci est particulièrement vrai en Python dont le typage dynamique repousse la découverte des fautes au moment de l'exécution.

1.1) Le traceback de Python

Lorsque l'interpréteur Python rencontre un problème, une exception est levée. Si elle n'est pas capturée, cette exception provoque l'arrêt du programme et l'affichage d'un message appelée "traceback". Ce message permet de connaître la nature et le contexte de l'incident.

Lire l'intégralité des messages d'erreur et se familiariser avec eux permet à la longue de gagner énormément de temps.

1.2) Les erreurs de syntaxe

Les erreurs de syntaxe empêchent l'interpréteur de comprendre le code écrit et provoque la levée d'une exception **avant même** l'exécution du code.

Ces erreurs sont souvent facile à trouver :

— parenthèse, crochet ou guillemet mal fermé (**SyntaxError**) ;

— mauvaise indentation (**IndentationError**)

Exemple :

```
for i in range(1,10):
    print("Quel est le carré de {}".format(i)
    print("C'est :{]".format(i**2))



File "/home/galand/Documents/Cours/NSI/Terminale/debog1.py", line 3
    print("C'est :{]".format(i**2))
          ^
SyntaxError: invalid syntax
```

L'erreur signalée ligne 3, est en fait ligne 2 : il manque une parenthèse fermante. Comme les instructions peuvent courir plusieurs lignes, elle n'est détectée que ligne 3.

1.3) Erreur à l'exécution

Les exceptions levées à l'exécution sont plus difficiles à trouver, car elles nécessitent de comprendre (à différents degrés) l'exécution du code. Elles sont également plus variées :

— **NameError :** un nom de variable a-t-il été mal orthographié ?

— **IndexError :** l'indice utilisé est-il en dehors de la liste ?

— **TypeError :** a-t-on essayé d'ajouter un nombre à une chaîne de caractères ?

Outre le type d'exception et la ligne l'ayant levée, le traceback contient un historique des appels de fonctions (la pile des appels) permettant de connaître le contexte d'exécution. La recherche d'une erreur s'apparente alors à une enquête : depuis l'endroit où l'erreur s'est déclarée, on remonte le fil d'exécution pour en déterminer la cause, qui peut être à un tout autre endroit (le traceback se lit pour cette raison vers le haut) :

```
Traceback (most recent call last) :
    File "error.py", line 10, in <module>
        f2()
    File "error.py", line 7, in f2
        f1()
    File "error.py", line 3, in f1
        a=a/(b+c)
```

Lors de l'exécution du fichier `error.py`, une erreur de type `ZeroDivisionError` a été levée, ligne 3 (a=a/(b+c)), dans la fonction f1(). La fonction en question a été exécutée suite à un appel sur la ligne 7, dans la fonction principale f2, qui elle même a été appelée par la ligne 10 du programme principal.

Remarque : Grace Hopper rapporte en 1947 un cas de bug devenu célèbre : il s'agissait d'un insecte (Bug en anglais) ayant provoqué des erreurs de calcul dans un ordinateur Mark II

1.4) Traitement des exceptions

Soit le programme suivant :

```
def affichage_inverse(lst):
    for x in lst:
        print(x,end='')
        print(1.0/x)

lst = [0.333, 2.5, 0, 10]
affichage_inverse(lst)
```

La console affiche les deux premiers résultats et nous indique l'erreur :

```
affichage_inverse
    print(1.0/x)

ZeroDivisionError: float division by zero
```

La dernière ligne du message d'erreur indique ce qui s'est passé. Dans notre exemple, le type de l'exception est ZeroDivisionError. On peut détecter les exceptions et réagir quand elle surviennent grâce aux instructions try/except :

```
def affichage_inverse(lst):
    for x in lst:
        print(x, end='')
        try:
            print(1.0 / x)
        except ZeroDivisionError:
            print("*** N admet pas d'inverse ***")

lst = [0.333, 2.5, 0, 10]
affichage_inverse(lst)
```

La console affiche correctement les inverses :

```
0.3333.003003003003003
2.50.4
0*** N admet pas d'inverse ***
100.1
```

Remarques : Quand on attrape une exception, le programme ne "plante pas". A la place de l'exception, le le bloc except (au nom de l'exception) qui est appelé.

Bien entendu, si une exception qui ne porte pas ce nom est levée, le programme plantera.

Une instruction **try** peut avoir plusieurs clauses d'exception, de façon à définir des gestionnaires d'exceptions différents pour des exceptions différentes.

```
def affichage_inverse(lst):
    for x in lst:
        print(x, end='')
        try:
            print(1.0 / x)
```

```
        except ZeroDivisionError:
            print("*** N admet pas d'inverse ***")
        except TypeError:
            print("*** Calcul d inverse impossible pour des types autre que int et float ***")

lst = [0.333, 2.5, 0, 10, "6"]
affichage_inverse(lst)
```

La console affiche les résultats sans plantage :

```
0.3333.003003003003003
2.50.4
0*** n'admet pas d'inverse ***
100.1
6***Calcul d'inverse impossible pour des types autre que int et float ***
```

On peut également provoquer soi-même des exceptions grâce à l'instruction `raise`. Prenons par exemple le cas où l'on souhaite traiter que des nombres positifs :

```
x=-1
if (x<0) :
    raise Exception("désolé, pas de nombre négatif !")
```

Dans ce cas, lever une exception n'est pas forcement la meilleure solution, on aurait pu utiliser un assert.

1.5) Débogueur

Le débogueur permet de dérouler un programme pas à pas et de vérifier l'état de chaque variable. C'est un outil parfois indispensable pour comprendre les bugs complexes.

La plupart des environnements de développement proposent un débogueur.

2) Documentation

Que l'on écrive ou qu'on utilise une fonction ou un module, la documentation est centrale pour que le travail soit réutilisable.

Parmi les différents mécanismes, l'un des plus simples est la **docstring** qui peut être rattachée à une fonction, à une procédure, à une méthode, à une classe, à un module ou à un package. Dans tous les cas, c'est une chaîne de caractères qui doit figurer au début de l'entité qu'elle documente.

Rappels : Fonctions, procédures, et classes peuvent être groupées par thèmes dans des fichiers séparés appelé **modules** qui peuvent être groupés en **packages**.

Cette documentation est ensuite consultable à l'aide de la commande `help` :

```
>>> import random
>>> help(random) #affiche la documentation du module
>>> help(print)  #affiche la documentation de la fonction
```

Exemple : écriture d'une docstring :

```python
def factorielle(n:int)-> int:
    """
    Calcul de la factorielle n! :
    n!=1*2*3*...*n

    >>>factorielle(5)
    120

    Parameters
    ----------
    n : int

    Returns
    -------
    int

    """
    res=1
    for i in range(2,n+1):
        res=res*1
    return res
```

3) Tests

Plutôt que de prouver la justesse d'un programme, on se contente de tester le code en vérifiant qu'il se comporte correctement sur des entrées pour lesquelles le résultat est connu.

3.1) Tests unitaires

De l'écriture des spécifications à la validation d'un logiciel, chaque étapes du développement est idéalement accompagné d'une phase de tests. Nous nous intéressons ici aux **tests unitaires dont l'objectif est de tester chaque fonction**.

Ces tests peuvent être conçus avant ou après la fonction à tester éventuellement par une personne différente. Ils sont généralement exécutés aussi souvent que possible pour éviter les problèmes de régression (ajout de bugs lors de modification du code).

Les tests unitaires doivent être le plus couvrant possible, c'est à dire envisager le plus de cas possibles (simples ou à problèmes). Chaque branche du code doit être idéalement testée.

3.2) Les outils de test

Il existe plusieurs outils permettant de réaliser des tests unitaires : les assertions, les doctests, le module tiers `pytest`...

Les exemples suivants concernent un fonction `fibo(n)` qui calcule le terme n de la suite de Fibonacci (suite définie par $F_0 = 0$, $F_1 = 1$ et pour $n > 1$, $F_n = F_{n-1} + F_{n-2}$).

a) Assertions

Une assertion échoue si l'expression booléenne qui suit le mot `assert` est fausse. Si elle est vraie, l'exécution continue sans erreur. Une assertion permet de vérifier,par exemple, le retour d'une fonction. Dès qu'une assertion est fausse, l'exécution s'arrête (exception **AssertionError**)

Exemple :

```
assert fibo(0)==0
assert fibo(1)==1
assert fibo(2)==1
```

b) Module doctest

Le module `doctest` permet d'intégrer des tests dans la docstring des fonctions. Les doctests sont repérés par la chaîne >>>. Écrire des doctests permet à la fois de réaliser des tests unitaires mais aussi de documenter efficacement la fonction.

.

```
def fibo(n):
    """
    calcule le nième terme de la suite de Fibonacci
    >>> fibo(3)
    2
    >>> fibo(7)
    13
    """
    fibo0=0
    fibo1=1
    for i in range(2, n+1):
        res=fibo0+fibo1+1
        fibo0=fibo1
        fibo1=res
    return res

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

Ce qui nous donne :

```
runfile('/Users/ericgaland/Documents/TNSI MAC/untitled0.py', wdir='/Users/ericgaland/Documents/TNSI MAC')
**********************************************************************
File "/Users/ericgaland/Documents/TNSI MAC/untitled0.py", line 12, in __main__.fibo
Failed example:
    fibo(3)
Expected:
    2
Got:
    4
**********************************************************************
File "/Users/ericgaland/Documents/TNSI MAC/untitled0.py", line 14, in __main__.fibo
Failed example:
    fibo(7)
Expected:
    13
Got:
    33
**********************************************************************
1 items had failures:
   2 of   2 in __main__.fibo
***Test Failed*** 2 failures.
```

c) Module pytest

Le module `pytest` permet de faire des tests plus complets (vérification qu'une exception est levée par exemple) et propose des diagnostics plus complets et plus explicites que les autres outils.

On peut démarrer l'utilisation de `pytest` par l'écritude de fonction préfixées par `test_` contenant chacune une assertion.

Exemple :

```
# Fichier test_fibo.py
import pytest
from fibo import fibo

def test_0():
    assert fibo(0)==0
def test_3():
    assert fibo(3)==2
def test_7():
    assert fibo(7)==13
def test_neg():
    with pytest.raises(ValueError):
        fibo(-1)
```

4) Mise en forme

On oublie souvent qu'un programme sera relu et modifié par quelqu'un d'autre : son auteur ou par un autre développeur. Adopter un style d'écriture standard facilite cette relecture.

4.1) Choisir correctement les noms

Syntaxiquement, les noms de variables, de fonctions , de classes, de méthodes, d'attributs, peuvent comporter des lettres, des chiffres, des caractères "_" et ne doivent pas commencer par un chiffre.

Quelque soit le langage, on choisira un nom d'autant plus évocateur qu'il est utilisé dans une grande portion de code.

La PEP 8 (guide de style Python accessible sur https://www.python.org/dev/peps/peps-008/ ajoute les conventions suivantes :

— nom de modules courts, en minuscules et de préférence sans "_". Exemple : `crypto`.

— nom de classes ou de types en CamelCase, c'est-à-dire en minuscules, sans "_" avec un majuscule en début de chaque mot. Exemple : `Class NombrePremier`.

— nom de fonctions, de méthodes, d'attributs ou de variables en minuscules, les mots séparés par des "_". Exemple : `def decomposition_facteurs_premiers(...)`

— les variables utilisés comme constantes sont en majuscules, les mots séparés par des "_". Exemple : `TOTAL_MAX=10`

4.2) Espaces, indentations, lignes blanches

a) Indentation

Les blocs Python sont délimités par l'indentation. La PEP 8 propose d'indenter les blocs à l'aide de 4 espaces (la plupart des éditeurs Python utilisent ce réglage).

Une ligne ne devrait pas excéder 79 caractères (cette règle est parfois transgressée). Lorsqu'une instruction court sur plusieurs lignes, on facilite la lecture en indentant. Exemple tiré de la PEP 8 :

```
#Aligner avec la parenthèse ouvrante
foo=nom_de _fonction_long(var_one, var_two,
                          var_three, var_four)
```

Enfin, on écrit une seule instruction par ligne.

b) Espaces

Les règles suivantes permettent de bien distribuer les espaces :

— pas d'espace avant le " :" ;
— espace après (mais pas avant) les "," dans les appels ou dans les définitions de fonctions ;
— espaces autour de "=" (pour l'affectation) et des opérateurs arithmétiques, sauf si il y en a beaucoup sur la ligne (dans ce cas, ne pas mettre d'espaces autour des plus prioritaires) ;
— pas d'espaces après "([", ni avant le ")]" ;
— pas d'espaces autour de " :" dans les *slices*.

Exemples

```
#espaces autour de =
a = 2
# espace après , et pas d'espace avant :
for i in range(1, 43):
    a = a * 2 # espace autour de = et de *
lst = [1, 1, 2, 2, 3, 5, 8] # espaces après ,
print(lst[2:6]) # pas d'espace autour de :
# espace après : et , mais pas avant
dico = {"6": 8, "7": 13, "8": 21}
```

c) Lignes blanches

On laisse deux lignes vides entre les différentes fonctions ou classes à l'intérieur d'un module ;
Au sein d'une classe, les méthodes sont séparées par une seule ligne blanche.

4.3) Outil de validation

Des outils sont souvent intégrés dans les environnements de développement, mais sont aussi disponibles séparément :

— `pep8` et`flake8` permettent de vérifier la conformité avec la PEP 8 (ainsi que d'autres choses).
— `autopep8` et`black` peuvent reformater le code en suivant ces règles.