CHAPITRE

1 Modularité et tests unitaires

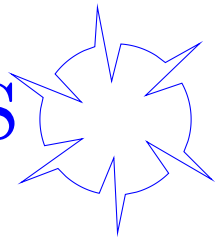

Table des matières

1 Structurer son code

Lorsqu'on écrit du code dans le développement d'un projet par exemple, il est important de respecter certaines règles syntaxiques comme placer des espaces entre les "=" ou après une virgule. La PEP8 recense les bonnes pratiques pour plus d'informations.
De la même façon, il devient nécessaire d'organiser son programme dès que le nombre de fonctions augmente. L'idée générale consiste en l'écriture d'un programme principal (le « main ») qui appelle des fonctions ou modules rangés dans des fichiers séparés : votre projet est ainsi plus lisible et compréhensible pour ceux qui voudraient le comprendre...

1.1 Paradigme de programmation

Il existe plusieurs façons de programmer et la **programmation fonctionnelle** en est une.

La programmation fonctionnelle repose sur l'utilisation de fonctions.

Vous aurez aussi l'occasion de découvrir en terminale la **programmation orientée objet** qui crée des objets avec des attributs et des méthodes qui modifient l'état de ces objets.
Le langage Python comme Javascript se prête volontiers à la programmation fonctionnelle. Notons que souvent les fonctions javascript sont des **procédures**, c'est-à-dire des fonctions qui modifient l'état d'un objet mais qui ne retournent pas de valeurs.

On parle d'**effets de bord** quand une fonction modifie l'état d'une variable en dehors de son environnement local. Il faut en général, les éviter !

Exemple n°1
La fonction ci-dessous crée un joli effet de bord, voulu ou non.

```
1 tab = [1, 2, 3, 4]
2 def incrementation_tab(t:list)-> None:
3     for i in range(len(t)):
4         t[i] += 1

5 incrementation_tab(tab)
6 print(tab)
```

Exercice n°1

❶ Expliquez pourquoi cette fonction crée un effet de bord.
❷ Que faudrait-il faire pour éviter cet effet ?

Les fonctions Python prennent en général un ou plusieurs **paramètres** en entrée, et retourne un résultat. Quelques exemples ci-dessous :

```python
#les fonctions de declarent avec la particule def
def calcul(nbre1, nbre2, op):
    if op == "somme":
        return nbre1 + nbre2
    elif op == "produit":
        return nbre1*nbre2
    elif op == "puissance":
        return nbre1**nbre2
    else:
        pass
```

La fonction prend en entrée trois paramètres et retourne soit un nombre soit rien... Les nouvelles déclarations de la PEP 484 permet de mieux renseigner la fonction :

```python
def calcul(nbre1:int, nbre2:int, op:str)->int:
    ...
```

> **Exercice n°2**
> Que donnent les instructions suivantes dans une console Python ?

```
>>> calcul(2, 3, somme)
>>> calcul(4, 5, "produit")
```

Il faut éviter les possibles confusions entre les variables déclarées et le nom des paramètres en évitant par exemple ce type de code :

```python
x = 4
def f(x):
    return x**2
```

Les deux "x" n'existent pas dans le même **espace de nommage** ; le premier est une **variable globale** disponible partout dans le programme principal le second est un paramètre de la fonction qui est **local**, c'est-à-dire seulement disponible au sein de la fonction.
Les variables créées dans une fonction sont locales !
Le programme principal ne peut pas lire la valeur d'une variable contenue dans une fonction.

```python
def f(x):
    x = x + 5
    return (x)

print(x)
```

1.2 L'importation de modules, fonctions, fichiers

La fonction suivante calcule le périmètre d'un cercle de rayon r. Que se passe t-il lors de l'appel de la fonction ?

```
1 def perim_cercle(r):
2     return 2*pi*r
3 perim_cercle(3)
```

```
>>> def perim_cercle(r):
...     return 2*pi*r
...
>>> perim_cercle(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in perim_cercle
NameError: name 'pi' is not defined
```

La fonction appelle la constante `pi` ; celle-ci existe dans la bibliothèque `math`. Il faut alors indiquer au programme où se trouve cette constante : c'est le principe de **l'importation**.On va chercher dans des modules pré-existants les fonctions ou constantes nécessaires. Il y a plusieurs façons d'importer une fonction ou un module :

```
1 from math import pi
2 def perim_cercle(r):
3     return 2*pi*r
```

```
1 from math import *
2 def perim_cercle(r):
3     return 2*pi*r
```

```
1 import math
2 def perim_cercle(r):
3     return 2*math.pi*r
```

Il faut éviter l'importation avec le joker $*$ car on ne sait pas vraiment ce qu'on importe : on pourrait même avoir des conflits sur le nom des fonctions car deux modules a priori différents peuvent contenir une fonction portant le même nom (comme `time` par exemple...).

Mais comment connaître le contenu d'un module ?

Facile tapez `help(math)` dans la console pour avoir toutes les fonctions du module math ! On peut aussi importer ces propres fichiers comme le montre l'exemple suivant.

Le fichier `pleindecalcul.py` présente en vrac plusieurs fonctions permettant le calcul de périmètre, d'aire ou de volume de certaines configurations. C'est, disons-le, un gros bordel !

```
######  pleindecalcul.py  #####
from math import pi
def aire_rect(L, l):
    return L*l
def perim_rect(L, l):
    return 2*(L+l)
def aire_cercle(r):
    return pi*r**2
def perim_cercle(r):
    return  2*pi*r
def aire_tri(L, h):
    return L*h/2
def vol_pave(L, l, h):
    return L*l*h
def vol_cylindre(r, h):
    return pi*r*r*h
def vol_cone(r, h):
    return pi*r*r*h/3
```

On propose une organisation plus rigoureuse de ces fonctions. Pour cela , on crée quatre fichiers : `perim.py`,`aire.py`,`volume.py` et `calculVAP.py`. Les trois premiers rassemblent les fonctions correspondantes et le dernier est le fichier principal qui **appelle** les autres fichiers par un import identique aux précédents :

```
####calculVAP.py#####
import perim
import aire
import volume

print(perim.perim_cercle(8))
print(volume.vol_pave(3,4,10))
```

On peut simplifier l'appel des fonctions dans leur fichier par la définition d'**alias** :

```
import perim as p
import aire as a
import volume as v

print(p.perim_cercle(8))
print(v.vol_pave(3,4,10))
```

1.3 Place des commentaires

Les commentaires sont des informations précieuses dans un programme : ils donnent des informations précieuses et/ou rendent inactifs certaines fonctions. Un commentaire en Python

débute par un # ; si vous voulez écrire plusieurs lignes, il faut alors utiliser le triple quote """ ... """.

1.4 Docstring des fonctions

La documentation des fonctions s'appelle le `docstring` : il indique ce que doit faire la fonction puis précise la nature des paramètres et ce que retourne la fonction. Pour plus d'informations consulter la PEP 257 qui ne parle que de ça...

```
def perim_cercle(r):
    """calcule le périmètre d'un cercle de rayon r
    param r : float positif
    sortie : périmètre du cercle de rayon r
    """
    return  2*pi*r
```

1.5 Le programme principal

Voici un exemple de fichiers python contenant une fonction qui agit sur les chaînes de caractères :

```
###### troncateur.py#######
def tronc(mot:str, n:int)->str:
    """garde les n premières lettres de la chaîne de caractères mot"""
    m = ""
    for i in range(n):
        m += mot[i]
    return m

if __name__ == "__main__":
    tronc("pharmacie", 4)
```

Si le fichier `troncateur.py` est exécuté il devient le programme principal(main), la condition `__name__ == "__main__"` est donc vraie et l'instruction `tronc("pharmacie", 4)` est donc exécutée !
En revanche, si ce fichier est importé dans un autre programme pour utiliser la fonction tronc, il n'est plus le programme principal et l'instruction ne sera pas exécutée !

2 Les conditions et les tests

2.1 Pré et post conditions

Vous aurez l'occasion de trouver dans les docstrings de fonction, des indications supplémentaires sur l'utilisation de celles-ci :

— des **préconditions**, qui indiquent quelles doivent être les conditions d'utilisations de la dite fonction , notamment sur les paramètres.

— des **postconditions**, qui garantissent la nature de la donnée retournée par la fonction. Ceci est un contrat qui lorsqu'il est respecté, garantie le bon fonctionnement de la fonction.

Ces conditions ont valeur de contrat entre celui qui code la fonction et celui qui l'utilise. Mais rien ne se passe si elles ne sont pas respectées. Les assertions(voir plus loin) régleront ce problème

2.2 Les erreurs ou exceptions

Les erreurs sont courantes en programmation même pour des professionnels. On distinguera cependant les **erreurs syntaxiques**(oubli de : ou mauvaises indentations sont des exemples courants...) des **erreurs logiques**. Python possède tout un catalogue d'erreurs et vous indique par une levée **d'une exception**, la caractère de cette erreur :

```
>>>10*(1/0)
>>>4 + spam*3
>>>'2' + 2
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Vous trouvez la liste des exceptions natives et leur signification dans Exceptions natives Python dans un bon moteur de recherche.

L'une des erreurs les plus levées est le fameux `index out of range`!

```
>>> tab = [2, 10, 5]
>>> print(tab[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Il est possible d'écrire des programmes qui prennent en charge certaines exceptions. Regardez l'exemple suivant, qui demande une saisie à l'utilisateur jusqu'à ce qu'un entier valide ait été entré, mais permet à l'utilisateur d'interrompre le programme (en utilisant Control-C ou un autre raccourci que le système accepte) ; notez qu'une interruption générée par l'utilisateur est signalée en levant l'exception KeyboardInterrupt. [1]

1. source :https ://docs.python.org/fr/3/tutorial/errors.html

```
while True:
    try:
        x =int(input("Entrer un nombre"))
        break
    except ValueError:
        print("Mauvais nombre, recommencez")
```

On peut même créer ses propres exceptions mais nous ne le développerons pas ici.

2.3 Assertions

Les conditions précisées précédemment ne garantissent pourtant pas que les règles d'utilisations soient respectées. Pour cela on utilise la fonction **assert** selon le principe suivant :si la condition n'est pas respectée, Python lève une exception, une erreur et affiche le type d'erreur déclaré dans l'assertion.

```
from math import sqrt
def racine_carre(x):
    assert x>=0, "x doit être positif"
    assert type(x) == float, "x doit être un nombre décimal"
    return sqrt(x)

racine_carre(-1)
racine_carre(a)
```

2.4 Doctest

Toujours dans la gestion des erreurs et des bugs, notamment des fonctions construites, il existe un moyen simple de les tester avant de les utiliser : ce sont **les tests unitaires**.

Quand vous écrivez une fonction, il faut la tester sur de nombreuses entrées, simples au départ.

Le principe est simple : dans le docstring on écrit ce que devrait donner le résultat de la fonction sur des entrées simples. L'importation du module `doctest` permet l'utilisation de sa méthode `testmod()` qui vérifie les tests.
Un exemple vaut mieux que de grands discours :

```python
import doctest
def pgcd(a,b):
    """
    pgcd(a,b): calcul du 'Plus Grand Commun Diviseur' \n
    entre les 2 nombres entiers a et b
    param : a et b deux entiers
    sortie: pgcd(a,b)
    >>> pgcd(15,10)
    5
    >>> pgcd(27,12)
    3
    """
    while b > 0:
        r = a%b
        a, b = b, r
    return a
doctest.testmod(verbose = True)
```

```
>>> %Run pgcd.py
Trying:
    pgcd(15,10)
Expecting:
    5
ok
Trying:
    pgcd(27,12)
Expecting:
    3
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.pgcd
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Si le test relève une erreur cela veut dire que votre fonction ne donne pas un résultat cohérent sur les entrées simples proposées : elle est sûrement mal codée (en considérant que les exemples sont justes...). Par exemple :

```python
from math import sqrt
from doctest import testmod

def distance(x, y):
    """...
    >>> distance(3,4)
    5
    """
    return sqrt(x*2 + y**2)
testmod()
```

```
>>> %Run testfaux.py
**********************************************************************
File "testfaux.py", line 6, in __main__.distance
Failed example:
    distance(3,4)
Expected:
    5
Got:
    4.69041575982343
**********************************************************************
1 items had failures:
   1 of   1 in __main__.distance
***Test Failed*** 1 failures.
```

Le mathématicien qui sommeille en vous a sûrement repéré l'erreur dans le calcul de la distance (il manque une * pour passer de la multiplication à la puissance...).

3 Exercices d'entraînement

Exercice n°3

On considère des chaînes de caractères contenant uniquement des majuscules et des caractères * appelées mots à trous. Par exemple `INFO*MA*IQUE`, `***I***E**` et `*S*` sont des mots à trous.

Écrire une fonction `correspond` qui :

- prend en paramètres deux chaînes de caractères `mot` et `mot_a_trous`
- retourne `True` si on peut obtenir `mot` en rempaçant convenablement les caractères `*` de `mot_a_trous` et `False` sinon

```
def correspond(mot:str, mot_a_trous:str)-> bool:
    ....
assert correspond('INFORMATIQUE', 'INFO*MA*IQUE') == True
assert correspond('AUTOMATIQUE', 'INFO*MA*IQUE') == False
assert correspond('STOP', 'S*') == False
assert correspond('AUTO', '*UT*') == True
```

Exercice n°4

Programmer la fonction `recherche`, prenant en paramètres un tableau non vide `tab` (type list) d'entiers et un entier `n`, et qui renvoie l'indice de la dernière occurrence de l'élément cherché. Si l'élément n'est pas présent, la fonction renvoie `None`.

```
assert recherche([2, 4], 2) == 0
assert recherche([1, 2, 1], 1) == 2
assert recherche([2, 4], 5) == None
```

Exercice n°5

La fonction `verifie_ordre` ci-dessous bien que correcte, est très maladroite. Expliquez pourquoi ?

```
def verifie_ordre(a:float, b:float)-> bool:
    if a < b:
        return True
    else:
        return False
```

Exercice n°6

Écrire une fonction `max_et_indice` qui prend en paramètre un tableau non vide `tab` de nombres entiers et qui renvoie sous forme de tuple, la valeur du plus grand élément de ce tableau ainsi que l'indice de sa première apparition dans ce tableau.

Exercice n°7

Ci-dessous, une fonction `count_vowels`

❶ Tester-là avec le module `doctest`.

❷ Le test marche t-il avec `count_vowels('Istanbul')` qui contient trois trois voyelles ?

❸ Améliorer cette fonction pour qu'elle prenne en compte les majuscules.

que je vous demande de tester avec le module `doctest`

```
def count_vowels(word):
    """
    Given a single word, return the total number of vowels in that single
    ↪  word.
    :param word: str
    :return: int
    >>> count_vowels('Cusco')
    2
    >>> count_vowels('Manila')
    3
    """
    total_vowels = 0
    for letter in word:
        if letter in 'aeiou':
            total_vowels += 1
    return total_vowels
```

Les fonctions **lambda** sont des fonctions anonymes en Python : elles n'ont en effet pas de nom. Elles peuvent être utilisées pour un besoin ponctuelle. Un exemple est donné dans le code ci-après.

```
def somme_classique(a, b):
    return a + b
#version lambda
sum_lambda = lambda a, b: a + b
sum_lambda(2, 4) # donne 6
```

Exercice n°8

Nous allons utiliser le module `turtle` de Python pour faire de la modularité.

❶ (a) Écrire une fonction `cercle` qui prend en paramètres un entier `r` pour le rayon et un tuple `couleur` pour les trois composantes `r,v,b` que vous enregistrez dans un fichier `trace_cercle.py`.

(b) Testez votre fonction sur plusieurs entrées (inutile de prendre `doctest`).

❷ De même écrire une fonction `rectangle` qui prend en paramètres deux entiers `h` et `l` pour les dimensions et un tuple `couleur` pour les trois composantes `r,v,b` que vous enregistrez dans un fichier `trace_rectangle.py`.

⚠ La construction commence au milieu de la base !

❸ Écrire une fonction `couleur_alea` sans paramètres, qui génère au hasard un tuple de trois valeurs entières entre 0 et 255 que vous enregistrerez

dans le fichier `gen_couleur.py`

❹ Écrire une fonction `triangle` qui prend en paramètres un entier `c` et un tuple `couleur` pour les trois composantes `r,v,b` que vous enregistrez dans un fichier `trace_triangle.py`. Cette fonction construit des triangles équilatéraux.

⚠ La construction commence au milieu de la base !

❺ Écrire le fichier principal `dessin.py` qui importe les trois fichiers précédents dans lequel vous construirez la figure ci-dessous (la boule a une couleur aléatoire...)