Spécifications et tests

1 Prototypage et typage

Prototyper une fonction, c'est :

- la nommer,
- définir ses paramètres,
- le type de variables utilisées,
- le type de la valeur retournée par la fonction.

Il n'est pas nécessaire de coder la fonction pour la prototyper. On peut utiliser le mot clé `pass` pour laisser un corps de fonction vide.

```
def puissance(facteur, exposant):
    pass
```

Il est de bonne pratique de nommer explicitement les fonctions et les paramètres, on doit comprendre leurs utilités juste avec leur nom. Les deux fonctions suivantes réalisent le même traitement, une est bien nommée.

```
def a(b, c):
    d = 1
    for _ in range(c):
        d = d * b
    return d
```

```
def puissance(facteur, exposant):
    resultat = 1
    for _ in range(exposant):
        resultat = resultat * facteur
    return resultat
```

Dans certains langages de programmation, une variable est définie avec un type qui n'est plus modifiable ensuite, sous peine d'erreur à la compilation. On appelle cela le **typage statique**.

```
// Ce code est écrit en langage C
int main()
{
    int a = 0;         // La variable a est définie comme un entier
    char a =  'Z';     // On essaie de la redéfinir en caractère
}
```

```
error: conflicting types for ‘a’; have ‘char’
      |     char a =  'Z';
      |          ^
note: previous definition of ‘a’ with type ‘int’
```

En Python, le **typage est dynamique**. Une variable possède un type (récupérable avec la fonction `type`) qui n'a pas besoin d'être explicité mais est vérifié lors de l'exécution d'opérations sur cette variable. Ce type est modifiable pendant l'exécution du code.

```
# On n'explicite jamais le type, Python le trouve de lui-même
a = 123
print(type(a))     # Type int
a = "Super"        # On change le type de a
print(type(a))     # Type str
```

Il est quand même possible de spécifier un type aux variables, aux paramètres et aux valeurs retournées par les fonctions. On appelle cela le **typage** (typing en anglais).

Voici un exemple de la même fonction sans et avec typage. Elle prend en paramètres deux nombres entiers et renvoie un nombre décimal. Chaque paramètre a son ou ses types de précisés au format `nom : type1|type2|...` , et le type de retour de la fonction aussi avec `-> type`.

```
# sans typage                 # avec typage
def moyenne2(n1, n2):         def moyenne2(n1 : int, n2 : int) -> float:
    return (n1 + n2) / 2          return (n1 + n2) / 2
```

Ce typage est non-contraignant, c'est à dire que rien ne nous oblige à le respecter en utilisant la fonction, cela ne provoque aucune erreur.

```
print(moyenne2(12, 14))      # respect du typage, renvoie 13.0
print(moyenne2(12.5, 14.5))  # non respect du typage, renvoie 13.5
```

Néanmoins, la définition et type et son respect permettent d'obtenir un **code plus robuste, avec moins de bugs**. Pour s'assurer du respect des types, il existe deux solutions :

- Vérifier les types des paramètres avec du code, avec des conditions ou des assertions (voir plus bas) ;
- Avec un **type checker** (hors programme), un outils de vérification de code. mypy est un type checker statique, c'est à dire qu'il lit (sans exécuter) le code pour vérifier qu'il n'y a pas d'erreur de non respect des types.

```
# Vérification des types fait avec une condition
def moyenne2(n1 : int, n2 : int) -> float:
    if type(n1) != int or type(n2) != int:
        print("Les paramètres doivent être entiers.")
        return -1
    return (n1 + n2) / 2

print(moyenne2(12.5, 14.5)) # Les paramètres doivent être entiers.
```

Voici une liste de quelques types fournis par Python : `int, float, str, bool, list, tuple`

2 Documentation

En plus du prototype et du typage d'une fonction, on souhaite généralement décrire le fonctionnement de notre fonction et comment l'utiliser correctement. On appelle cela **documenter son code**.

On peut déjà faire cela avec de simples commentaires dans le code quand celui-ci reste à petite échelle, mais il existe un outils plus puissant et plus adapté aux gros projets: la **chaîne de documentation** (docstring en anglais).
La chaîne de documentation s'écrit dans le bloc de la fonction, juste en dessous de sa définition, en utilisant 3 guillemets doubles : `""" ... """`.

```
def somme(a : int|float, b : int|float) -> int|float:
    """ Calcule a + b, avec a et b des nombres entiers ou décimaux.
    """
    return a + b
```

On peut récupérer la chaîne de documentation en utilisant la fonction `help()` dans la console.

```
help(somme)
Help on function somme in module __main__:

somme(a: int | float, b: int | float) -> int | float
    Calcule a + b, avec a et b des nombres entiers ou décimaux.
```

Python fournit déjà une documentation pour ses fonctions et donne en plus des informations sur les paramètres et les erreurs qui peuvent être levées.

```
help(list.pop)
Help on method_descriptor:

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

Cette chaîne de documentation peut aussi contenir du code de tests (voir plus bas).
A l'aide d'outils annexes, il est possibles de générer toute la documentation d'un projet sous la forme d'un document imprimable ou d'un site web.

3 Jeux de tests

Il est **indispensable de tester son code** après l'avoir écrit pour s'assurer qu'il fonctionne, mais certaines méthodes de développement, comme le TDD (Test driven development) préconisent même que l'on écrive un code de test avant d'avoir coder une fonctionnalité.

L'ajout d'une fonctionnalité avec la méthode de développement logiciel TDD se déroule ainsi :

1. On commence par écrire le test.
2. Le test échoue (forcément, rien n'est encore codé !)
3. On écrit le code pour que le test soit validé.
4. On améliore (si possible) ce code tout en vérifiant que le test continue à être valide.

On répète ces étapes pour chaque nouvelle fonctionnalité, en relançant les précédents tests pour s'assurer de ne rien avoir involontaire cassé avec du nouveau code. On appelle cela des tests de non-régression.

Les **tests ne prouveront jamais qu'un code ne contient pas d'erreurs**, ils sont néanmoins utiles et il est important de bien les penser pour qu'ils couvrent un maximum de cas possibles. Par exemple, si on prend le cas d'une fonction qui recherche la valeur maximale dans une liste d'entiers, des cas de tests possibles seraient :

- On passe une liste vide.
- On passe une liste avec des entiers, des entiers positifs, négatifs, les deux...
- On passe une liste avec le maximum à différentes positions, une fois ou plusieurs fois...

Les **assertions** sont des tests utilisés pour prévenir des situations qui seraient logiquement impossibles. Si une de ces situations est détectée, alors le programme n'est pas sûr ou est mal codé et il est arrêté. De manière plus concrète, il existe deux cas possibles :

- On veut repérer si des données (souvent des paramètres d'une fonction) sont susceptibles de créer des problèmes, ou sont hors spécification : on appelle cela des **préconditions sur les arguments**.
- On veut repérer si les fonctions que l'on a écrit se comportent correctement et renvoie bien les résultats attendus : on appelle cela des **postconditions sur les résultats**.

On utilise le mot clé `assert` avec la syntaxe : *`assert test, message_si_test_vaut_False`*

```
def somme(a : int|float, b : int|float) -> int|float:
    """ Calcule a + b, avec a et b des nombres entiers ou décimaux.
    """
    # tests les préconditions
    assert type(a) == int or type(a) == float , "a doit être un nombre."
    assert type(b) == int or type(b) == float , "b doit être un nombre."

    return a + b

# tests les postconditions
assert somme(1, 1) == 2
assert somme(-0.125, -0.5) == -0.625
```

Les assertion nous permettent de remplacer un test avec une conditionnelle pour détecter des erreurs.
Elles offrent d'autres avantages (hors programme):

- On peut choisir de les désactiver lors de l'exécution du programme pour gagner en performance.
- Il est possible de gérer les erreurs par le mécanisme `try except`, qui permet de lever des exceptions.

Il est possible de les intégrer des tests dans la chaîne de documentation et de les exécuter grâce au module `doctest` et sa fonction `testmod`. Dans la chaîne de documentation, les appels à la fonction sont précédées de >>> et le résultat attendu placé à la ligne qui suit.

```
def somme(a : int|float, b : int|float) -> int|float:
    """ Calcule a + b, avec a et b des nombres entiers ou décimaux.
    >>> somme(1, 1)
    2
    >>> somme(-0.125, -0.5)
    -0.625
    """
    return a + b


>>> import doctest
>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

On peut aussi demander à la fonction d'afficher en détail les tests effectués avec l'appel `doctest.testmod(verbose=True)`.

```
>>> import doctest
>>> doctest.testmod(verbose=True)
Trying:
    somme(1, 1)
Expecting:
    2
ok
Trying:
    somme(-0.125, -0.5)
Expecting:
    -0.625
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.somme
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=2)
```