Spécification et tests

Prototypes, docstrings, préconditions, postconditions, tests et bibliothèques

Dans ce chapitre, nous allons découvrir comment documenter correctement nos fonctions Python grâce aux **prototypes**, aux **docstrings** et aux **tests**.

1 Le prototype d'une fonction

Définition 1 **(Prototype de fonction)** Le **prototype** d'une fonction est la première ligne de sa définition. Il précise le nom de la fonction, les paramètres avec leurs types, et le type de retour.

Exemple 1. `def aire_rectangle(longueur: float, largeur: float) -> float:`
Cette fonction prend deux paramètres de type `float` et retourne un `float`.

2 La docstring

Définition 2 **(Docstring)** Une **docstring** est une chaîne de documentation placée après le prototype. Elle documente ce que fait la fonction, ses paramètres et sa valeur de retour.

Exemple 2 (Exemple de docstring). Voici la fonction `aire_rectangle` complétée avec une docstring.

```
def aire_rectangle(longueur: float, largeur: float) -> float:
    """Calcule l'aire d'un rectangle.
    Paramètres :
        longueur (float) : la longueur du rectangle.
        largeur (float) : la largeur du rectangle.
    Renvoie :
        float : l'aire du rectangle (longueur * largeur).
    """
    return longueur * largeur
```

Structure d'une docstring. Une docstring complète contient :

1. Une description brève
2. Les paramètres avec leurs types (section `Paramètres :`)
3. La valeur de retour et ce que la fonction garantit sur elle, c'est-à-dire la postcondition (section `Renvoie :`)
4. Des exemples (doctests, section `Exemples :`)
5. Les préconditions si nécessaire (section `Préconditions :`)

③ Les préconditions

Définition 3 **(Précondition)** Une **précondition** est une condition que doivent vérifier les paramètres pour que la fonction s'exécute correctement.

Exemple 3. Pour une fonction `calculer_racine_carree(x: float) -> float`, une précondition est que `x` doit être positif ou nul. Pour une fonction `diviser(a: float, b: float) -> float`, une précondition est que `b` doit être différent de 0.

Vérifier avec assert. On utilise l'instruction `assert` pour vérifier les préconditions : `assert condition, "message d'erreur"`

Exemple 4 (Utilisation de assert).

```
def calculer_racine_carree(x: float) -> float:
    """Calcule la racine carrée d'un nombre."""
    assert x >= 0, "La racine carrée n'est définie que pour les nombres positifs."
    return x ** 0.5
```

Si on appelle `calculer_racine_carree(-4)`, le programme s'arrêtera et affichera :

```
AssertionError: La racine carrée n'est définie que pour les nombres positifs.
```

④ Les postconditions

Définition 4 **(Postcondition)** Une **postcondition** est une condition que vérifie le résultat de la fonction lorsque les préconditions sont respectées. Elle décrit ce que la fonction garantit sur la valeur qu'elle renvoie ; dans la docstring, elle figure dans la section `Renvoie :`.

Exemple 5. La fonction `note_sur_20` convertit des points en note sur 20. Si le total est strictement positif et si les points sont compris entre 0 et ce total (préconditions), la note renvoyée est comprise entre 0 et 20 (postcondition).

Vérifier avec assert. On vérifie une postcondition avec l'instruction `assert`, juste avant le `return`.

Exemple 6 (Précondition et postcondition).

```python
def note_sur_20(points: float, total: float) -> float:
    """Convertit des points en note sur 20.
    Paramètres :
        points (float) : les points obtenus.
        total (float) : le total des points possibles.
    Préconditions : total > 0 et 0 <= points <= total.
    Renvoie :
        float : la note sur 20, comprise entre 0 et 20 (postcondition).
    """
    assert total > 0 and 0 <= points <= total, "Données incorrectes."
    note = points / total * 20
    assert 0 <= note <= 20, "La note doit être comprise entre 0 et 20."
    return note
```

Si l'on se trompe dans la formule, par exemple en écrivant `note = total / points * 20`, l'appel `note_sur_20(15.0, 30.0)` calcule `40.0` et le programme s'arrête :

```
AssertionError: La note doit être comprise entre 0 et 20.
```

L'erreur est repérée dans la fonction elle-même, avant que la note fausse ne soit utilisée ailleurs dans le programme.

5 Les doctests

Définition 5 **(Doctest)** Un **doctest** est un exemple d'utilisation inclus dans la docstring. Il commence par >>> suivi de l'instruction et du résultat attendu.

Exemple 7 (Exemple de doctests).

```python
def aire_rectangle(longueur: float, largeur: float) -> float:
    """Calcule l'aire d'un rectangle.
    Paramètres :
        longueur (float) : la longueur du rectangle.
        largeur (float) : la largeur du rectangle.
    Renvoie :
        float : l'aire du rectangle.
    Exemples :
        >>> aire_rectangle(10.0, 5.0)
        50.0
        >>> aire_rectangle(7.0, 3.0)
        21.0
        >>> aire_rectangle(0.0, 10.0)
        0.0
    """
    assert longueur >= 0 and largeur >= 0, "Les dimensions doivent être positives."
    return longueur * largeur
```

Méthode : exécuter les doctests. Pour exécuter les doctests, on ajoute le bloc ci-dessous à la fin du fichier Python.

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

6 Les jeux de tests

Définition 6 **(Jeu de tests)** Un **jeu de tests** est un ensemble d'appels de la fonction dont on connaît à l'avance le résultat attendu. Les doctests en forment un ; on peut aussi l'écrire avec des instructions `assert`, une par appel.

```python
assert aire_rectangle(10.0, 5.0) == 50.0   # cas courant
assert aire_rectangle(7.0, 3.0) == 21.0    # cas courant
assert aire_rectangle(0.0, 10.0) == 0.0    # cas limite
```

Méthode : choisir les tests. La qualité des tests compte autant que leur nombre : on teste des cas courants, mais aussi des **cas limites**, c'est-à-dire des valeurs extrêmes (une valeur nulle, une liste vide, la borne d'un intervalle).

Limite des tests. Le succès d'un jeu de tests ne garantit pas la correction d'un programme. Par exemple, la version fautive `return longueur * longueur` réussit le test du cas limite, car 0×0 et 0×10 valent tous deux 0 : avec ce seul test, l'erreur passerait inaperçue. Un jeu de tests peut révéler des erreurs, mais jamais prouver qu'il n'y en a aucune.

7 Utilisation de bibliothèques

Définition 7 **(Bibliothèque)** Une **bibliothèque** (ou module) est un fichier Python contenant des fonctions réutilisables comme `math`, `random`, etc.

Exemple 8. Si vous avez besoin de calculer une racine carrée ou de générer un nombre aléatoire, vous n'avez pas besoin de réécrire ces fonctions. Vous utiliserez respectivement les bibliothèques `math` et `random`.

Méthode : importer une bibliothèque. Syntaxes possibles :

- `import math` (importe tout le module)
- `from math import sqrt` (importe seulement la fonction `sqrt`)
- `import math as m` (importe le module en lui donnant un surnom)

Exemple 9 (Utilisation d'une fonction importée).

```python
# Utilisation 1 : import math
import math
print(math.sqrt(16)) # Affiche 4.0
print(math.pi)       # Affiche 3.14159...

# Utilisation 2 : from math import sqrt, pi
from math import sqrt, pi
print(sqrt(25)) # Affiche 5.0
print(pi)       # Affiche 3.14159...

# Utilisation 3 : import math as m
import math as m
print(m.sqrt(36)) # Affiche 6.0
```

⑧ Synthèse

Résumé. Une fonction Python de qualité professionnelle contient :

1. un prototype clair avec annotations de types ;
2. une docstring complète (description, paramètres, renvoie, exemples) ;
3. une vérification des préconditions avec `assert` au début de la fonction, et de la postcondition avec `assert` avant le `return` ;
4. des doctests variés (cas normaux, cas limites), en sachant que la réussite d'un jeu de tests ne prouve pas que la fonction est correcte.