NSI · TERMINALE / COURS

Modularité et mise au point

Astrolab Cours · astrolabcours.fr

01 Le problème

Un programme de 3 000 lignes dans un seul fichier, sans fonctions : impossible de trouver une erreur, de le partager entre plusieurs développeurs ou d'en réutiliser un morceau. Comment construire un gros programme qui reste fiable ?

01 L'idée : les briques standard

Avec des briques de construction aux attaches standard, tu assembles une maison sans savoir comment chaque brique a été fabriquée. Il suffit que chacune respecte sa forme. En programmation, une brique est une fonction ou un module, et sa forme, c'est sa **spécification**.

01 Découper en fonctions et en modules

- Une **fonction** fait une seule tâche précise.
- Un **module** est un fichier `.py` qui regroupe des fonctions sur un même thème.
- On l'utilise avec `import`.

```
# fichier geometrie.py
from math import pi

def aire_disque(r):
    """Renvoie l'aire d'un disque de rayon r (r >= 0)."""
    return pi * r ** 2
```

```
# fichier principal
import geometrie
print(geometrie.aire_disque(2))
```

L'ensemble des fonctions qu'un module met à disposition, avec leur documentation, forme son **API** (interface de programmation). On peut s'en servir sans lire son code.

Spécifier

La **docstring** (entre triples guillemets, juste sous `def`) dit ce que fait la fonction, ce qu'elle attend (**préconditions**) et ce qu'elle garantit (**postconditions**).

© Astrolab Cours · astrolabcours.fr/nsi/terminale/modularite-et-mise-au-point/ · Tous droits réservés : usage personnel uniquement, republication et usage commercial interdits

Tester

```
def moyenne(t):
    """Renvoie la moyenne d'une liste non vide de nombres."""
    assert len(t) > 0, "liste vide"
    return sum(t) / len(t)

assert moyenne([10, 20]) == 15
assert moyenne([7]) == 7
```

`assert condition` ne fait rien si la condition est vraie, et arrête le programme avec une `AssertionError` sinon. On écrit les tests avant ou pendant le code, en pensant aux cas limites.

Gérer les erreurs

Quand une erreur est prévisible (saisie incorrecte, fichier absent), on la traite au lieu de laisser le programme planter :

```
try:
    age = int(input("Âge : "))
except ValueError:
    print("Il faut un nombre entier.")
```

Exemple guidé

Écrire une fonction `moyenne` spécifiée, testée, et qui signale clairement une mauvaise utilisation.

```
def moyenne(notes):
    """Renvoie la moyenne d'une liste non vide de nombres."""
    if len(notes) == 0:
        raise ValueError("liste vide")
    return sum(notes) / len(notes)

assert moyenne([10, 20]) == 15
assert moyenne([12]) == 12
```

1. **La docstring** dit ce que la fonction attend et ce qu'elle renvoie : quelqu'un d'autre peut l'utiliser sans lire le code.
2. **Les assertions** testent un cas normal et un cas limite (une seule note). Si on modifie la fonction plus tard, elles signalent tout de suite une régression.
3. **Le cas interdit** (liste vide) lève une erreur explicite, plutôt qu'un `ZeroDivisionError` obscur au milieu d'un autre calcul.

Rangée dans un fichier `stats.py`, la fonction s'importe ailleurs avec `from stats import moyenne`.

Retour au problème

Les 3 000 lignes deviennent des modules de fonctions courtes, chacune spécifiée et testée. Une erreur se

© Astrolab Cours · astrolabcours.fr/nsi/terminale/modularite-et-mise-au-point/ · Tous droits réservés : usage personnel uniquement, republication et usage commercial interdits

cherche dans une fonction précise, et chaque développeur travaille sur son module.

Les pièges

- Ne tester que des cas « normaux » : les erreurs se cachent dans les cas limites.
- Comparer des flottants avec `==` dans un test : `0.1 + 0.2 == 0.3` est faux.
- Un `except` sans type d'erreur attrape tout, même les fautes de frappe : préciser l'erreur attendue.

© Astrolab Cours · astrolabcours.fr/nsi/terminale/modularite-et-mise-au-point/ · Tous droits réservés : usage personnel uniquement, republication et usage commercial interdits