**PLAN DE TEST – Projet Super Calculatrice**

**FICHIER : operators.py**

**Méthodes publiques (à tester directement)**

| **Méthode** | **Typage** | **Cas à tester** |
| --- | --- | --- |
| addition | `def addition(self, operation: str) -> float | None:` |
| substraction | `def substraction(self, operation: str) -> float | None:` |
| multiplication | `def multiplication(self, operation: str) -> float | None:` |
| division | `def division(self, operation: str) -> float | None:` |

| **Cas spécifique** | **Exemple(s)** | **Pourquoi le tester ?** |
| --- | --- | --- |
| Nombres négatifs | "-2+5" | Pour s'assurer que le signe est bien interprété |
| Espaces aléatoires | " 3 + 5 " | Tester la tolérance aux espaces multiples |
| Très grands nombres | "1e10+1e10" | Vérifier les limites de précision |
| Format scientifique | "1.2e3+1" | Tester le support des notations scientifiques |
| Résultat à virgule | "1/3" | Vérifier la précision du résultat |
| Résultat zéro | "2-2", "0\*10" | Vérifier le bon traitement du zéro |
| Division négative | "-8/2" | Résultat attendu : -4.0 |
| Mauvaise position de symbole | "\*2+2", "2+2+" | Ne doit pas être interprété comme valide |

**Méthodes privées :**

* **Utiliser unittest.mock ou tests ciblés pour forcer l’appel de \_is\_symbol\_valid() ou \_is\_float()**

**Cas d’erreur à tester (toutes méthodes) :**

| **Cas** | **Exemple(s)** | **Résultat attendu** |
| --- | --- | --- |
| Symboles incorrects | "3#2", "2 & 3" | None |
| Caractères non numériques | "5+a" | None |
| Mélange d’opérateurs | "3 + 4 - 2" (si dans addition) | None |
| Chaînes vides / espaces | "", " " | None |
| Doublons d’opérateurs | "3++2", "3--1" | None |
| Opération avec un seul nombre | "5" | 5.0 ou None (selon logique définie) |

**Méthodes privées (testées indirectement)**

* \_is\_operation\_valid()
* \_is\_symbol\_valid()
* \_is\_float(value)
* \_calculate\_addition/substraction/...

Ces méthodes seront testées uniquement via les appels publics.

**FICHIER : view.py**

| **Méthode** | | **Typage** | **Test à réaliser** | |
| --- | --- | --- | --- | --- |
| print\_menu() | | def print\_menu() -> None: | Mock de print(), vérifier contenu affiché | |
| get\_user\_input(input\_msg: str) -> str | | Simuler input() avec pytest-mock ou unittest.mock.patch |  | |
| end\_message() | | def end\_message() -> None: | Mock print(), vérifier "GOOD-BYE" | |
| continue\_message() | | def continue\_message() -> None: | Mock input(), s'assurer qu'il attend l'entrée | |
| `print\_result(operation: str, result: float | | None) -> None` | ✔️ "3+2", 5.0" → affiche "= 5.0" ❌ "a+2", None" → affiche erreur | |
| get\_user\_input() vide | Simuler un utilisateur qui appuie directement sur Entrée | | |

|  |  |
| --- | --- |
| print\_result() avec texte non opérable | Ex : "4#2" et None → tester que le message d’erreur est bien affiché |

|  |  |
| --- | --- |
| print\_result() avec float précis | "1/3" → 0.333333..., vérifier l'affichage avec arrondi éventuellement |

**FICHIER : controller.py**

| **Méthode** | **Typage** | **Test à réaliser** |
| --- | --- | --- |
| run() | def run(self) -> None: | Simuler des saisies utilisateur via mocks (input) pour tester navigation menu |
| \_is\_input\_valid(user\_input: str) -> bool | "1" → True "5" → False "abc" → False |  |
| \_operations(user\_input: str) -> None | Vérifier que : - "1" → addition - "4" → division - Appelle bien View.print\_result() - Input invalide ne déclenche pas de traitement |  |
| \_is\_quit(user\_input: str) -> bool | "5" → False (quitte), "1" → True (continue) |  |

| **Cas spécifique** | **Pourquoi le tester ?** |
| --- | --- |
| Enchaînement de plusieurs opérations | "1", "2+2", "3", "3\*2", "5" |
| Tester invalidité persistante | "xyz", "@", "", "6", "0" → doivent être ignorés |
| Test de sortie immédiate | Lancer "5" directement → doit quitter sans erreur |
| Surcharge d'input | Simuler 10 inputs pour vérifier que le programme reste stable |

**Tests de boucle :**

* "1" + "2+2" + "5" → affiche résultat, puis quitte
* "abc" → rien ne se passe
* "1" + "2+b" → affiche erreur

**FICHIER : main.py**

| **Test** | **Commentaire** |
| --- | --- |
| Test indirect par appel à Controller().run() | Pas de test unitaire dédié requis Inclus via les tests controller.py |

**NOUVELLES MÉTHODES – Tests à ajouter (scientifiques/statistiques/visualisation)**

FICHIER : operators.py

Tests unitaires à ajouter pour les méthodes suivantes :

🔬 Scientifiques :

- power("2 ^ 3") → 8

- square\_root("sqrt(16)") → 4, avec cas d’erreur sur sqrt(-1)

- logarithm("log(1)") → 0, avec cas d’erreur sur log(0) et log(-x)

- modulo("5 % 2") → 1, avec cas d’erreur sur %0

- sine("sin(0)") → 0, idem pour cos et tan avec valeur mathématique correcte

- factorial("5!") → 120, erreurs si négatif ou décimal

- absolute("abs(-5)") → 5

- exponential("exp(1)") → e

📊 Statistiques :

- mean("mean(1,2,3,4,5)") → 3

- median("median(1,2,3,4,5)") → 3

- mode("mode(1,2,2,3,3,3)") → 3

- standard\_deviation("std(1,2,3,4,5)")

- variance("var(1,2,3,4,5)")

- percentile("percentile(1,2,3,4,5;50)") → 3

- correlation("correlation(1,2,3;4,5,6)") → 1

- linear\_regression("regression(1,2,3;4,5,6)") → dict avec slope, intercept, r²

📈 Visualisation (retourne un texte confirmant la sauvegarde de l'image) :

- plot\_function("plot(x^2, -10, 10)")

- scatter\_plot("scatter(1,2,3,4)")

- histogram("histogram(1,2,2,3,3,3)")

- polar\_plot("polar(2\*sin(theta), 0, 2\*pi)")

Chaque fonction est testée en cas normal ET en cas d’erreur (type invalide, valeur incorrecte, etc.).