An Introduction to Unit Testing

With Examples in Python (pytest) and JavaScript (Jest)

1 What Is a Unit Test?

A *unit test* is a small piece of code that:

- exercises a small part of your code (a function, method, or class),
- checks that the result matches the expected behaviour,
- can be executed automatically, usually together with many other tests.

Typical properties of unit tests:

- **Small scope:** they focus on a single behaviour.
- **Deterministic:** same inputs, same outputs.
- **Fast:** hundreds or thousands should run in seconds.
- **Isolated:** they avoid external dependencies (databases, APIs, global state).

2 The Arrange–Act–Assert Pattern

Most unit tests follow the *Arrange–Act–Assert* (AAA) structure:

1. **Arrange:** set up the data and objects required for the test.
2. **Act:** call the function or method you want to test.
3. **Assert:** check that the actual result matches the expected result.

Conceptually, a test looks like:

Listing 1: Generic Arrange–Act–Assert structure

```
def test_something() -> None:
    # Arrange
    input_data = ...
    expected = ...

    # Act
    result = my_function(input_data)

    # Assert
    assert result == expected
```

If any assertion fails, the test fails.

3 Python Example: Testing a Mean Function with `pytest`

Consider a small module `math_utils.py`:

Listing 2: Python function to compute a mean

```
# math_utils.py
from typing import Iterable


```

```python
def mean(values: Iterable[float]) -> float:
    """
    Compute the arithmetic mean of an iterable of floats.

    Args:
        values: Iterable of numeric values.

    Returns:
        The arithmetic mean as a float.

    Raises:
        ValueError: If 'values' is empty.
    """
    # Convert to list so we can iterate multiple times
    values_list = list(values)

    if not values_list:
        raise ValueError("mean() requires at least one value.")

    total: float = sum(values_list)      # Sum of all values
    count: int = len(values_list)        # Number of values

    return total / count
```

We can test this function using `pytest` in a file `test_math_utils.py`:

Listing 3: Unit tests for the `mean` function using pytest

```python
# test_math_utils.py
import pytest

from math_utils import mean


def test_mean_with_positive_numbers() -> None:
    """
    'mean' should return the correct value for a simple list
    of positive numbers.
    """
    # Arrange
    values = [1.0, 2.0, 3.0, 4.0]
    expected: float = 2.5

    # Act
    result: float = mean(values)

    # Assert
    assert result == expected


def test_mean_with_negative_numbers() -> None:
    """
    'mean' should handle negative numbers correctly.
    """
    # Arrange
    values = [-1.0, -3.0]
    expected: float = -2.0

    # Act
```

```
    result: float = mean(values)

    # Assert
    assert result == expected


def test_mean_raises_error_on_empty_input() -> None:
    """
    'mean' should raise ValueError when given an empty iterable.
    """
    # Arrange
    values: list[float] = []

    # Act / Assert
    with pytest.raises(ValueError):
        mean(values)
```

Running `pytest` from the project root will discover and execute all functions whose names start with `test_`.

4 JavaScript Example: Testing a Mean Function with Jest

Now consider a similar implementation in JavaScript, using Jest for testing.

Production Code (`mathUtils.js`)

Listing 4: JavaScript function to compute a mean

```
/**
 * Compute the arithmetic mean of an array of numbers.
 *
 * @param {number[]} values - Array of numeric values.
 * @returns {number} The arithmetic mean.
 * @throws {Error} If 'values' is empty.
 */
function mean(values) {
  // Ensure we are working with an array
  if (!Array.isArray(values)) {
    throw new Error("mean() requires an array of numbers.");
  }

  if (values.length === 0) {
    throw new Error("mean() requires at least one value.");
  }

  // Compute the sum of the values
  const total = values.reduce((accumulator, value) => {
    if (typeof value !== "number") {
      throw new Error("All elements must be numbers.");
    }
    return accumulator + value;
  }, 0);

  const count = values.length; // Number of elements

  return total / count;
}
```

```
30
31 // Export for testing
32 module.exports = { mean };
```

Unit Tests with Jest (`mathUtils.test.js`)

Listing 5: Unit tests for the JavaScript `mean` function using Jest

```
1  const { mean } = require("./mathUtils");
2
3  describe("mean", () => {
4    test("returns correct mean for positive numbers", () => {
5      // Arrange
6      const values = [1.0, 2.0, 3.0, 4.0];
7      const expected = 2.5;
8
9      // Act
10     const result = mean(values);
11
12     // Assert
13     expect(result).toBe(expected);
14   });
15
16   test("returns correct mean for negative numbers", () => {
17     // Arrange
18     const values = [-1.0, -3.0];
19     const expected = -2.0;
20
21     // Act
22     const result = mean(values);
23
24     // Assert
25     expect(result).toBe(expected);
26   });
27
28   test("throws error on empty array", () => {
29     // Arrange
30     const values = [];
31
32     // Act / Assert
33     expect(() => mean(values)).toThrow("mean() requires at least one
           value.");
34   });
35
36   test("throws error when elements are not numbers", () => {
37     // Arrange
38     const values = [1, "two", 3];
39
40     // Act / Assert
41     expect(() => mean(values)).toThrow("All elements must be numbers.");
42   });
43 });
```

With Jest installed (for example via `npm install -save-dev jest`), you can run:

```
1 npx jest
```

from the project root.

5 A Class Example: Bank Account

5.1 Python Version

Production Code (`bank.py`)

Listing 6: Simple bank account class in Python

```python
# bank.py
from __future__ import annotations

from dataclasses import dataclass


@dataclass
class BankAccount:
    """
    Simple bank account model with deposit and withdraw operations.
    """
    owner: str
    balance: float = 0.0

    def deposit(self, amount: float) -> None:
        """
        Deposit a positive amount into the account.

        Args:
            amount: Positive amount to add to the balance.

        Raises:
            ValueError: If 'amount' is not positive.
        """
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")

        # Increase balance by the given amount
        self.balance += amount

    def withdraw(self, amount: float) -> None:
        """
        Withdraw a positive amount from the account if sufficient funds exist.

        Args:
            amount: Positive amount to withdraw.

        Raises:
            ValueError: If 'amount' is not positive or exceeds the balance.
        """
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")

        if amount > self.balance:
            raise ValueError("Insufficient funds.")

        # Decrease balance by the given amount
        self.balance -= amount
```

Unit Tests (`test_bank.py`)

Listing 7: Unit tests for the Python `BankAccount` class

```python
# test_bank.py
import pytest

from bank import BankAccount


def test_deposit_increases_balance() -> None:
    """
    Depositing a positive amount should increase the account balance.
    """
    # Arrange
    account = BankAccount(owner="Alice", balance=100.0)

    # Act
    account.deposit(50.0)

    # Assert
    assert account.balance == 150.0


def test_deposit_negative_amount_raises_error() -> None:
    """
    Depositing a non-positive amount should raise ValueError.
    """
    # Arrange
    account = BankAccount(owner="Bob", balance=100.0)

    # Act / Assert
    with pytest.raises(ValueError):
        account.deposit(-10.0)


def test_withdraw_decreases_balance() -> None:
    """
    Withdrawing a valid amount should decrease the balance.
    """
    # Arrange
    account = BankAccount(owner="Carol", balance=200.0)

    # Act
    account.withdraw(80.0)

    # Assert
    assert account.balance == 120.0


def test_withdraw_more_than_balance_raises_error() -> None:
    """
    Withdrawing more than the balance should raise ValueError.
    """
    # Arrange
    account = BankAccount(owner="Dave", balance=50.0)

    # Act / Assert
    with pytest.raises(ValueError):
```

```
        account.withdraw(60.0)
```

5.2 JavaScript Version

Production Code (`bankAccount.js`)

Listing 8: Simple bank account class in JavaScript

```
/**
 * Simple bank account model with deposit and withdraw operations.
 */
class BankAccount {
  /**
   * @param {string} owner - Name of the account owner.
   * @param {number} [initialBalance=0] - Initial account balance.
   */
  constructor(owner, initialBalance = 0) {
    if (typeof owner !== "string" || owner.length === 0) {
      throw new Error("Owner name must be a non-empty string.");
    }

    if (typeof initialBalance !== "number" || Number.isNaN(
        initialBalance)) {
      throw new Error("Initial balance must be a valid number.");
    }

    this.owner = owner;
    this.balance = initialBalance;
  }

  /**
   * Deposit a positive amount into the account.
   *
   * @param {number} amount - Positive amount to deposit.
   */
  deposit(amount) {
    if (typeof amount !== "number" || amount <= 0) {
      throw new Error("Deposit amount must be a positive number.");
    }

    // Increase the balance
    this.balance += amount;
  }

  /**
   * Withdraw a positive amount from the account if sufficient funds
       exist.
   *
   * @param {number} amount - Positive amount to withdraw.
   */
  withdraw(amount) {
    if (typeof amount !== "number" || amount <= 0) {
      throw new Error("Withdrawal amount must be a positive number.");
    }

    if (amount > this.balance) {
      throw new Error("Insufficient funds.");
    }
```

```
49
50      // Decrease the balance
51      this.balance -= amount;
52    }
53  }
54
55  module.exports = { BankAccount };
```

Unit Tests with Jest (`bankAccount.test.js`)

Listing 9: Unit tests for the JavaScript `BankAccount` class

```
1  const { BankAccount } = require("./bankAccount");
2
3  describe("BankAccount", () => {
4    test("deposit increases balance", () => {
5      // Arrange
6      const account = new BankAccount("Alice", 100);
7
8      // Act
9      account.deposit(50);
10
11     // Assert
12     expect(account.balance).toBe(150);
13   });
14
15   test("depositing non-positive amount throws", () => {
16     // Arrange
17     const account = new BankAccount("Bob", 100);
18
19     // Act / Assert
20     expect(() => account.deposit(0)).toThrow(
21       "Deposit amount must be a positive number."
22     );
23     expect(() => account.deposit(-10)).toThrow(
24       "Deposit amount must be a positive number."
25     );
26   });
27
28   test("withdraw decreases balance", () => {
29     // Arrange
30     const account = new BankAccount("Carol", 200);
31
32     // Act
33     account.withdraw(80);
34
35     // Assert
36     expect(account.balance).toBe(120);
37   });
38
39   test("withdrawing more than balance throws", () => {
40     // Arrange
41     const account = new BankAccount("Dave", 50);
42
43     // Act / Assert
44     expect(() => account.withdraw(60)).toThrow("Insufficient funds.");
45   });
46 });
```

6 What Makes a Good Unit Test?

Some practical guidelines:

- **One behaviour per test:** avoid mixing unrelated checks in a single test.
- **Clear names:** e.g. `test_mean_raises_error_on_empty_input`.
- **No hidden magic:** tests should be easy to read and understand.
- **Fast and independent:** tests should not depend on execution order or external state.
- **Test behaviour, not implementation details:** if internal code changes but behaviour stays the same, tests should still pass.

7 Minimal Project Structures

For a small Python project:

Listing 10: Example Python project structure

```
my_project/
  my_package/
    __init__.py
    math_utils.py
    bank.py
  tests/
    test_math_utils.py
    test_bank.py
```

Run:

```
pytest
```

For a small JavaScript project using Jest:

Listing 11: Example JavaScript project structure

```
my-js-project/
  mathUtils.js
  bankAccount.js
  mathUtils.test.js
  bankAccount.test.js
  package.json
```

A minimal `package.json`:

```
{
  "name": "my-js-project",
  "version": "1.0.0",
  "devDependencies": {
    "jest": "^29.0.0"
  },
  "scripts": {
    "test": "jest"
  }
}
```

Then run:

```
npm install
npm test
```