# `unittest` in Python

## 1. What Is `unittest`?

`unittest` is Python's built-in testing framework.

It allows you to write automated tests for your Python code without installing an external package.

Unlike `pytest`, which must be installed separately, `unittest` comes with Python.

```
import unittest
```

You can use it to:

- organize tests
- run multiple test cases
- make assertions
- test exceptions
- set up test data
- clean up after tests
- group related tests
- run tests automatically

---

# 2. Why Does `unittest` Exist?

Imagine you create this function:

```
def add(a, b):
    return a + b
```

You could manually check it:

```
print(add(2, 3))
print(add(10, 5))
print(add(-1, 1))
```

But manual checking becomes difficult as a project grows.

With `unittest`, you can create repeatable automated checks:

```
class TestAdd(unittest.TestCase):

    def test_add_two_numbers(self):
        self.assertEqual(add(2, 3), 5)
```

Then Python can run the test for you.

---

## 3. `unittest` vs `pytest`

Both are used for Python testing.

| Feature | `unittest` | `pytest` |
|---|---|---|
| Included with Python | Yes | No |
| Installation required | No | Yes |
| Main style | Class based | Function based |
| Assertions | `self.assertEqual()` | `assert` |
| Fixtures | `setUp()` / `tearDown()` | Fixtures |
| Parameterization | More verbose | Convenient |
| Learning curve | Structured | Generally simpler |
| Third party | No | Yes |

Neither framework changes the fundamental purpose of testing.

The important concepts remain:

```
Arrange
   ↓
```

```
Act
   ↓
Assert
```

---

# 4. Importing `unittest`

Because it is part of Python's standard library:

```
import unittest
```

No:

```
pip install unittest
```

is required.

This is an important distinction between standard-library modules and third-party packages.

---

# 5. Your First `unittest` Test

Suppose you have:

```
def add(a, b):
    return a + b
```

Create:

```
test_calculator.py
```

Then:

```
import unittest


def add(a, b):
    return a + b


class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)


if __name__ == "__main__":
    unittest.main()
```

Run:

```
python test_calculator.py
```

If the test passes, `unittest` reports that the test succeeded.

---

## 6. The `TestCase` Class

The central building block of `unittest` is:

```
unittest.TestCase
```

You normally create a class that inherits from it:

```
class TestCalculator(unittest.TestCase):
    ...
```

This gives your test class access to methods such as:

```
self.assertEqual()
self.assertTrue()
self.assertFalse()
self.assertIsNone()
self.assertIn()
```

and many others.

---

## 7. Test Methods

Test methods normally begin with:

```
test_
```

Example:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        ...

    def test_subtract(self):
        ...

    def test_multiply(self):
        ...
```

The naming convention allows `unittest` to discover these methods automatically.

A method named:

```
check_add()
```

will not normally be treated as a test.

A method named:

```
test_add()
```

will.

---

## 8. `assertEqual()`

The most common assertion is:

```
self.assertEqual(actual, expected)
```

Example:

```
self.assertEqual(add(2, 3), 5)
```

This checks:

```
actual == expected
```

Another example:

```
self.assertEqual("Python".upper(), "PYTHON")
```

---

## 9. `assertNotEqual()`

Use:

```
self.assertNotEqual(actual, unexpected)
```

Example:

```
self.assertNotEqual(add(2, 3), 10)
```

The test passes because:

```
5 != 10
```

---

## 10. Boolean Assertions

### `assertTrue()`

```
self.assertTrue(is_even(4))
```

### `assertFalse()`

```
self.assertFalse(is_even(5))
```

Example:

```
def is_even(number):
    return number % 2 == 0
```

Tests:

```
class TestNumbers(unittest.TestCase):

    def test_even_number(self):
        self.assertTrue(is_even(4))

    def test_odd_number(self):
        self.assertFalse(is_even(5))
```

---

## 11. Testing `None`

Use:

```
self.assertIsNone(value)
```

Example:

```
def find_user(users, user_id):
    for user in users:
        if user["id"] == user_id:
            return user

    return None
```

Test:

```
class TestUsers(unittest.TestCase):

    def test_missing_user(self):
        users = [
            {"id": 1, "name": "Hafsa"}
        ]

        result = find_user(users, 99)
```

```
        self.assertIsNone(result)
```

---

## 12. Testing That Something Is Not `None`

Use:

```
self.assertIsNotNone(value)
```

Example:

```
result = find_user(users, 1)

self.assertIsNotNone(result)
```

This checks that a value exists.

---

## 13. Testing Membership

Use:

```
self.assertIn(member, container)
```

Example:

```
languages = ["Python", "JavaScript", "Dart"]

self.assertIn("Python", languages)
```

For absence:

```
self.assertNotIn("Java", languages)
```

---

## 14. Testing Object Identity

Sometimes you need to check whether two variables refer to the exact same object.

Use:

```
self.assertIs(first, second)
```

Example:

```
value = None

self.assertIs(value, None)
```

For different objects:

```
self.assertIsNot(first, second)
```

This is different from:

```
assert first == second
```

because equality and identity are different concepts.

---

# 15. Testing Types

You can check an object's type with:

```
self.assertIsInstance(value, expected_type)
```

Example:

```
result = add(2, 3)

self.assertIsInstance(result, int)
```

You can also check subclasses through normal Python type relationships.

---

# 16. Testing Exceptions

Testing errors is an important part of testing.

Suppose:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")

    return a / b
```

With `unittest`:

```
class TestDivide(unittest.TestCase):

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)
```

The test passes when the expected exception is raised.

---

# 17. Checking the Exception Message

You can capture the exception:

```
class TestDivide(unittest.TestCase):

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError) as context:
            divide(10, 0)

        self.assertEqual(
            str(context.exception),
            "Cannot divide by zero"
        )
```

This checks both:

```
Exception type
+
Exception message
```

Use message assertions when the message itself is part of the behavior you need to guarantee.

---

# 18. Testing Strings

Example:

```
def get_username():
    return "Hafsa"
```

Test:

```
class TestUser(unittest.TestCase):

    def test_username(self):
        self.assertEqual(get_username(), "Hafsa")
```

You can also use:

```
self.assertIn("Haf", get_username())
```

---

# 19. Testing Lists

Example:

```
def get_languages():
    return ["Python", "JavaScript", "Dart"]
```

Test exact contents:

```
self.assertEqual(
    get_languages(),
    ["Python", "JavaScript", "Dart"]
)
```

Test membership:

```
self.assertIn("Python", get_languages())
```

Test length:

```
self.assertEqual(len(get_languages()), 3)
```

---

# 20. Testing Dictionaries

Example:

```
def get_user():
    return {
        "name": "Hafsa",
        "role": "developer"
    }
```

Test:

```
class TestUser(unittest.TestCase):

    def test_user(self):
        user = get_user()

        self.assertEqual(user["name"], "Hafsa")
        self.assertEqual(user["role"], "developer")
```

You can also test:

```
self.assertIn("name", user)
```

---

# 21. Multiple Tests in One Class

A test class can contain many related tests.

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

```
    def test_subtract(self):
        self.assertEqual(subtract(5, 2), 3)

    def test_multiply(self):
        self.assertEqual(multiply(4, 3), 12)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
```

Grouping related behavior makes the test suite easier to navigate.

---

## 22. The `setUp()` Method

Sometimes multiple tests need the same initial data.

Instead of repeating that setup:

```
class TestUsers(unittest.TestCase):

    def setUp(self):
        self.users = [
            {"id": 1, "name": "Hafsa"},
            {"id": 2, "name": "Asim"},
        ]
```

Then each test can access:

```
self.users
```

Example:

```
class TestUsers(unittest.TestCase):

    def setUp(self):
        self.users = [
            {"id": 1, "name": "Hafsa"},
            {"id": 2, "name": "Asim"},
        ]

    def test_first_user(self):
        self.assertEqual(self.users[0]["name"], "Hafsa")

    def test_user_count(self):
        self.assertEqual(len(self.users), 2)
```

---

## 23. How `setUp()` Works

`setUp()` runs before every test method.

For:

```
def test_one(self):
    ...


def test_two(self):
    ...
```

the sequence is approximately:

```
setUp()
   ↓
test_one()
   ↓
setUp()
   ↓
test_two()
```

This gives each test fresh setup.

That helps keep tests independent.

---

## 24. The `tearDown()` Method

`tearDown()` runs after each test.

It is useful for cleanup.

```
class TestFiles(unittest.TestCase):

    def setUp(self):
        ...

    def tearDown(self):
        ...
```

The general sequence is:

```
setUp()
   ↓
test
```

```
    ↓
tearDown()
```

For example, if a test creates a temporary resource, `tearDown()` can clean it up.

---

## 25. `setUp()` and `tearDown()` Mental Model

Think:

```
Prepare
  ↓
Run test
  ↓
Clean up
```

This structure is especially useful when tests involve:

- temporary files
- database connections
- test objects
- temporary directories
- external resources

---

## 26. `setUpClass()` and `tearDownClass()`

Sometimes setup should happen once for the entire test class.

Use:

```
@classmethod
def setUpClass(cls):
    ...
```

and:

```
@classmethod
def tearDownClass(cls):
    ...
```

Example:

```
class TestDatabase(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
```

```
        cls.connection = create_connection()

    @classmethod
    def tearDownClass(cls):
        cls.connection.close()
```

The approximate lifecycle is:

```
setUpClass()
   ↓
test 1
   ↓
test 2
   ↓
test 3
   ↓
tearDownClass()
```

Use this carefully because shared state can make tests less isolated.

---

## 27. `setUp()` vs `setUpClass()`

| Method | Runs |
|---|---|
| `setUp()` | Before every test |
| `tearDown()` | After every test |
| `setUpClass()` | Once before all tests in a class |
| `tearDownClass()` | Once after all tests in a class |

For most simple tests, `setUp()` is easier and safer.

---

## 28. Organizing Tests in Files

A project can look like:

```
project/
│
├── app/
│   ├── calculator.py
│   └── resources.py
│
├── tests/
│   ├── test_calculator.py
│   └── test_resources.py
│
└── README.md
```

Example:

```
app/calculator.py
```

```
def add(a, b):
    return a + b
```

Then:

```
tests/test_calculator.py
```

```
import unittest

from app.calculator import add


class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

---

## 29. Running `unittest`

You can run an individual test file:

```
python -m unittest tests/test_calculator.py
```

You can also run test discovery:

```
python -m unittest discover
```

This searches for tests according to `unittest`'s discovery rules.

---

# 30. Test Discovery

A common convention is:

```
test_*.py
```

For example:

```
test_calculator.py
test_users.py
test_resources.py
```

Inside those files, methods beginning with:

```
test
```

can be discovered as tests.

Run:

```
python -m unittest discover
```

to discover and execute the test suite.

---

# 31. Verbose Test Output

For more detailed output:

```
python -m unittest -v
```

The `-v` option means verbose.

This is useful when you want to see individual test names rather than only a compact summary.

---

# 32. Running a Specific Test

You can target a particular module or test.

For example:

```
python -m unittest tests.test_calculator
```

Or a specific test method:

```
python -m unittest tests.test_calculator.TestCalculator.test_add
```

This is useful when debugging a failing test without running the entire suite.

---

## 33. `subTest()`

Sometimes you want to test multiple inputs while keeping them within one test method.

`subTest()` lets you identify each case separately.

```
class TestNumbers(unittest.TestCase):

    def test_is_even(self):
        cases = [
            (2, True),
            (4, True),
            (5, False),
            (7, False),
        ]

        for number, expected in cases:
            with self.subTest(number=number):
                self.assertEqual(is_even(number), expected)
```

If one case fails, the output identifies the relevant subtest.

---

## 34. Testing Boundaries

Suppose:

```
def is_adult(age):
    return age >= 18
```

A useful test checks the boundary:

```
class TestAge(unittest.TestCase):

    def test_is_adult(self):
        self.assertFalse(is_adult(17))
        self.assertTrue(is_adult(18))
        self.assertTrue(is_adult(19))
```

The important value is:

```
18
```

because behavior changes at that point.

---

# 35. Testing Invalid Inputs

Suppose:

```
def validate_username(username):
    if len(username) < 3:
        raise ValueError("Username is too short")

    return True
```

Tests:

```
class TestUsername(unittest.TestCase):

    def test_valid_username(self):
        self.assertTrue(validate_username("Hafsa"))

    def test_short_username(self):
        with self.assertRaises(ValueError):
            validate_username("Hi")
```

Good test suites protect both successful and unsuccessful behavior.

---

# 36. Testing Behavior Instead of Implementation

Suppose your function is:

```
def add(a, b):
    return a + b
```

A useful test is:

```
self.assertEqual(add(2, 3), 5)
```

The test does not care whether the implementation uses an intermediate variable:

```
def add(a, b):
    result = a + b
    return result
```

The behavior remains the same.

Good tests allow implementation details to change without unnecessarily breaking the test suite.

---

# 37. Test Independence

Avoid:

```
test_create_user
       ↓
test_update_user
       ↓
test_delete_user
```

where each test assumes another test has already run.

Instead, each test should prepare its own state.

```
def test_update_user(self):
    user = create_test_user()

    update_user(user)

    self.assertEqual(user["name"], "New Name")
```

This makes tests easier to run independently.

---

# 38. Testing a haas.dev Resource Service

Imagine a simplified haas.dev resource service:

```
def filter_resources(resources, category):
    return [
        resource
        for resource in resources
        if resource["category"] == category
    ]
```

A `unittest` test could be:

```
import unittest


class TestResourceService(unittest.TestCase):

    def setUp(self):
        self.resources = [
            {
                "title": "Python Basics",
                "category": "Python"
            },
```

```
            {
                "title": "React Basics",
                "category": "React"
            },
            {
                "title": "Python Testing",
                "category": "Python"
            },
        ]

    def test_filter_python_resources(self):
        result = filter_resources(self.resources, "Python")

        self.assertEqual(len(result), 2)
        self.assertEqual(result[0]["title"], "Python Basics")
        self.assertEqual(result[1]["title"], "Python Testing")
```

This test protects the resource filtering behavior.

---

# 39. Testing Empty Results

The same service should also handle categories with no matches.

```
def test_filter_missing_category(self):
    result = filter_resources(
        self.resources,
        "Flutter"
    )

    self.assertEqual(result, [])
```

This protects an important edge case.

---

# 40. Testing Empty Input

Another useful test:

```
def test_empty_resource_list(self):
    result = filter_resources([], "Python")

    self.assertEqual(result, [])
```

This makes the expected behavior explicit.

---

## 41. Testing Multiple Cases With `subTest()`

For the same resource service:

```
def test_filter_categories(self):
    cases = [
        ("Python", 2),
        ("React", 1),
        ("Flutter", 0),
    ]

    for category, expected_count in cases:
        with self.subTest(category=category):
            result = filter_resources(
                self.resources,
                category
            )

            self.assertEqual(len(result), expected_count)
```

This keeps related cases together while preserving useful failure information.

---

## 42. Common `unittest` Assertions

| Assertion | Purpose |
|---|---|
| `assertEqual(a, b)` | `a == b` |
| `assertNotEqual(a, b)` | `a != b` |
| `assertTrue(x)` | `x` is truthy |
| `assertFalse(x)` | `x` is falsy |
| `assertIs(a, b)` | Same object |

| | |
|---|---|
| `assertIsNot(a, b)` | Different objects |
| `assertIsNone(x)` | `x is None` |
| `assertIsNotNone(x)` | `x is not None` |
| `assertIn(a, b)` | `a in b` |
| `assertNotIn(a, b)` | `a not in b` |
| `assertIsInstance(a, b)` | `isinstance(a, b)` |
| `assertRaises(E)` | Expected exception |

These cover many everyday testing situations.

---

# 43. Common Mistakes

## Mistake 1: Forgetting `TestCase`

Incorrect:

```
class TestCalculator:
    ...
```

Correct:

```
class TestCalculator(unittest.TestCase):
    ...
```

---

## Mistake 2: Forgetting the `test_` Prefix

Incorrect:

```
def add_test(self):
    ...
```

Better:

```
def test_add(self):
    ...
```

---

## Mistake 3: Using the Wrong Assertion

Instead of:

```
self.assertTrue(result == 5)
```

prefer:

```
self.assertEqual(result, 5)
```

The second version communicates the intent more clearly.

---

## Mistake 4: Making Tests Dependent

Tests should generally be independent.

---

## Mistake 5: Sharing Mutable State

Be careful when using class-level or global mutable data.

Fresh setup per test is often safer.

---

## Mistake 6: Testing Only Successful Cases

Also test:

```
invalid input
empty input
boundary values
expected exceptions
```

where relevant.

---

## Mistake 7: Ignoring Failed Tests

A failing test is useful evidence.

Investigate whether:

```
the implementation is wrong
```

or:

```
  the test expectation is wrong
```

before changing anything.

---

# 44. `unittest` and Mocking

`unittest` also includes a mocking system:

```
  from unittest.mock import Mock
```

Mocks are useful when your code depends on external systems.

For example:

```
  Application
      ↓
  Email Service
```

A test does not necessarily need to send a real email.

Instead, a mock can represent the email service.

A simple example:

```
  from unittest.mock import Mock

  email_service = Mock()

  email_service.send("Hafsa")

  email_service.send.assert_called_once_with("Hafsa")
```

This verifies that the interaction occurred without calling a real service.

Mocking is an important topic on its own and should be studied separately from basic `unittest` fundamentals.

---

# 45. `unittest.mock` Mental Model

Without a mock:

```
  Your code
     ↓
  Real API
     ↓
  Network
```

```
   ↓
External service
```

With a mock:

```
Your code
   ↓
Mock
   ↓
Controlled test behavior
```

This makes certain tests:

- faster
- isolated
- predictable

But excessive mocking can make tests fragile, so mocks should be used where they provide a clear testing benefit.

---

# 46. Test Fixtures in `unittest`

A fixture is the setup required for a test.

In `unittest`, common fixture mechanisms include:

```
setUp()
tearDown()
setUpClass()
tearDownClass()
```

Think of them as lifecycle hooks.

```
Test class starts
       ↓
setUpClass()
       ↓
setUp()
       ↓
test
       ↓
tearDown()
       ↓
setUp()
       ↓
```

```
test
       ↓
tearDown()
       ↓
tearDownClass()
```

Understanding this lifecycle is important when tests use resources.

---

# 47. A Complete `unittest` Example

Application:

```
def calculate_total(price, quantity):
    if price < 0:
        raise ValueError("Price cannot be negative")

    if quantity < 0:
        raise ValueError("Quantity cannot be negative")

    return price * quantity
```

Tests:

```
import unittest

from shop import calculate_total


class TestCalculateTotal(unittest.TestCase):

    def test_calculate_total(self):
        self.assertEqual(
            calculate_total(10, 3),
            30
        )

    def test_zero_quantity(self):
        self.assertEqual(
            calculate_total(10, 0),
            0
        )

    def test_negative_price(self):
        with self.assertRaises(ValueError):
```

```
            calculate_total(-10, 2)

    def test_negative_quantity(self):
        with self.assertRaises(ValueError):
            calculate_total(10, -2)


if __name__ == "__main__":
    unittest.main()
```

This small suite covers:

```
normal input
zero boundary
invalid price
invalid quantity
```

---

# 48. A Practical `unittest` Workflow

When adding a feature:

### Step 1

Define the expected behavior.

### Step 2

Identify:

```
normal cases
edge cases
invalid cases
exceptions
boundaries
```

### Step 3

Create or update a `TestCase`.

### Step 4

Write focused test methods.

### Step 5

Run:

```
python -m unittest discover
```

### Step 6

Read failures carefully.

### Step 7

Fix the underlying problem.

### Step 8

Run the complete suite again.

### Step 9

Refactor the implementation while keeping tests green.

---

# 49. Mini Project

Build a small resource manager.

Create:

```
class ResourceManager:

    def __init__(self):
        self.resources = []

    def add(self, resource):
        self.resources.append(resource)

    def remove(self, resource_id):
        self.resources = [
            resource
            for resource in self.resources
            if resource["id"] != resource_id
        ]

    def find(self, resource_id):
        for resource in self.resources:
            if resource["id"] == resource_id:
                return resource

        return None
```

Create a test class:

```
import unittest


class TestResourceManager(unittest.TestCase):
```

```
    def setUp(self):
        self.manager = ResourceManager()

        self.manager.add({
            "id": 1,
            "title": "Python"
        })

        self.manager.add({
            "id": 2,
            "title": "Testing"
        })
```

Then write tests for:

1. Adding a resource
2. Finding an existing resource
3. Finding a missing resource
4. Removing a resource
5. Removing a missing resource
6. Resource count
7. Empty manager behavior

---

# 50. 5 Minute Challenge

Create:

```
def is_valid_email(email):
    ...
```

For this exercise, define a simple rule:

```
The string must contain @
```

Write a `unittest.TestCase` containing tests for:

```
"user@example.com" → True
"hello" → False
"@example.com" → True or False based on your chosen rule
"" → False
```

The important part is not the complexity of email validation.

The goal is to practice:

```
TestCase
test methods
assertTrue
assertFalse
```

---

# 51. Exercises

## Exercise 1

Create:

```
def is_positive(number):
    ...
```

Write tests for:

```
10
1
0
-1
-10
```

---

## Exercise 2

Create:

```
def get_longest_word(words):
    ...
```

Test:

```
["Python", "JavaScript", "Go"]
["Python"]
[]
```

Decide how an empty list should behave.

---

## Exercise 3

Create:

```
def validate_password(password):
    ...
```

Require at least 8 characters.

Test:

```
valid password
7 characters
8 characters
empty password
```

---

## Exercise 4

Create:

```
def calculate_shipping(total):
    ...
```

Use:

```
total < 50 → shipping fee
total >= 50 → free shipping
```

Test:

```
49
50
51
```

---

## Exercise 5

Create a `ResourceManager` and test:

```
add()
find()
remove()
```

Use `setUp()` to create fresh test data before every test.

---

# 52. Mini Quiz

### 1. Is `unittest` included with Python?

A. No

B. Yes

C. Only on Linux

D. Only with Anaconda

**Answer: B**

**2. What should a test method normally begin with?**

A. `check_`

B. `run_`

C. `test_`

D. `verify_`

**Answer: C**

**3. Which class should test classes usually inherit from?**

A. `Test`

B. `unittest.TestCase`

C. `unittest.Test`

D. `PythonTest`

**Answer: B**

**4. Which assertion checks equality?**

A. `assertSame()`

B. `assertEqual()`

C. `assertEqualsTo()`

D. `assertMatch()`

**Answer: B**

**5. Which method runs before every test?**

A. `before()`

B. `prepare()`

C. `setUp()`

D. `initialize()`

**Answer: C**

**6. Which method is commonly used to test an expected exception?**

A. `assertError()`

B. `assertRaises()`

C. `assertException()`

D. `assertFail()`

**Answer: B**

---

# 53. Interview Questions

### Beginner

1. What is `unittest`?
2. Why use `unittest`?
3. Is `unittest` included in Python?
4. What is `unittest.TestCase`?
5. How does `unittest` discover test methods?
6. What is `assertEqual()`?
7. What is `setUp()`?
8. What is `tearDown()`?
9. How do you run `unittest` tests?
10. What does `python -m unittest discover` do?

### Intermediate

11. What is the difference between `setUp()` and `setUpClass()`?
12. What is `subTest()`?
13. How do you test exceptions with `unittest`?
14. How do you test whether two objects are identical?
15. How do you test collection membership?
16. Why should tests be independent?
17. What is the purpose of `unittest.mock`?
18. What is the difference between `assertEqual()` and `assertIs()`?
19. Why should tests focus on behavior rather than implementation details?
20. What are the advantages of using a test fixture?

### Practical

21. How would you test a function with invalid input?
22. How would you test a database-dependent function?
23. How would you test a function that sends an email?
24. How would you organize tests in a large Python project?
25. How would you investigate a failing `unittest` test?

---

## 54. `unittest` Cheat Sheet

### Import

```
import unittest
```

### Test class

```
class TestSomething(unittest.TestCase):
    ...
```

### Test method

```
def test_something(self):
    ...
```

### Equality

```
self.assertEqual(actual, expected)
```

### Inequality

```
self.assertNotEqual(actual, expected)
```

### True

```
self.assertTrue(value)
```

### False

```
self.assertFalse(value)
```

### None

```
self.assertIsNone(value)
```

### Not None

```
self.assertIsNotNone(value)
```

### Membership

```
self.assertIn(value, collection)
```

### Not in

```
self.assertNotIn(value, collection)
```

### Type

```
self.assertIsInstance(value, SomeClass)
```

### Exception

```
with self.assertRaises(ValueError):
    function()
```

### Setup

```
def setUp(self):
    ...
```

### Cleanup

```
def tearDown(self):
    ...
```

### Run a file

```
python -m unittest tests/test_file.py
```

### Discover tests

```
python -m unittest discover
```

### Verbose mode

```
python -m unittest -v
```

### Run a specific test

```
python -m unittest tests.test_file.TestClass.test_method
```

---

# 55. Key Takeaways

- `unittest` is Python's built-in testing framework.
- It requires no separate installation.
- Tests are usually organized inside `unittest.TestCase` classes.
- Test methods normally start with `test_`.
- Assertions verify expected behavior.
- `assertEqual()` checks equality.
- `assertTrue()` and `assertFalse()` check Boolean behavior.
- `assertRaises()` verifies expected exceptions.
- `setUp()` prepares fresh state before every test.
- `tearDown()` performs cleanup after every test.
- `setUpClass()` and `tearDownClass()` operate at the class level.

- `subTest()` is useful for testing multiple related cases.
- `unittest.mock` helps isolate code from external dependencies.
- Tests should remain focused, readable, independent, and deterministic.
- Testing edge cases is as important as testing the happy path.
- Test coverage measures exercised code, not necessarily test quality.
- Tests should protect behavior rather than unnecessary implementation details.
- `unittest` can scale from simple function tests to larger application test suites.

The core pattern is:

```
Arrange
   ↓
Act
   ↓
Assert
```

And the core goal is:

```
Write code
   ↓
Define expected behavior
   ↓
Automate the check
   ↓
Run tests
   ↓
Fix failures
   ↓
Refactor safely
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources