Testing Fundamentals in Python

1. What Is Software Testing?

Software testing is the process of checking whether a program behaves as expected.

When you write code, you usually have an idea of what should happen:

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

You expect:

```
add(2, 3)
```

to return:

```
5
```

A test automatically checks that expectation.

```
assert add(2, 3) == 5
```

If the result is `5`, the test passes.

If the result is something else, the test fails.

Testing is not about proving that software has no bugs.

It is about creating reliable checks that help you detect incorrect behavior early.

---

2. Why Testing Matters

Without tests, developers often check code manually.

For example:

```
def calculate_total(price, quantity):
    return price * quantity
```

You might manually try:

```
print(calculate_total(10, 3))
print(calculate_total(5, 2))
print(calculate_total(100, 4))
```

This works for a few cases.

But imagine a project with 50 functions.

Every time you change the code, manually checking everything becomes slow and unreliable.

Tests give you repeatable checks.

Instead of remembering what to test, you can run:

```
pytest
```

and let the test suite check the application.

The basic idea

```
Write code
   ↓
Write tests
   ↓
Run tests
   ↓
Pass?
  ┌┴┐
Yes  No
 ↓    ↓
Continue
      ↓
Investigate
      ↓
Fix
      ↓
Run tests again
```

---

3. Testing vs Debugging

Testing and debugging are related but different.

Testing

Testing asks:

> Does this code behave correctly?

Debugging

Debugging asks:

> Why is this code behaving incorrectly?

For example:

```
def divide(a, b):
    return a * b
```

A test can reveal the problem:

```
assert divide(10, 2) == 5
```

The test fails.

Debugging then helps you discover that the implementation uses `*` instead of `/`.

Simple mental model

```
Testing → Finds evidence of a problem

Debugging → Finds the cause of the problem
```

---

4. What Makes a Good Test?

A useful test should be:

Specific

It should test one clear behavior.

Bad:

```
def test_everything():
    ...
```

Better:

```
def test_add_returns_sum():
    ...
```

Repeatable

Running the test multiple times should normally produce the same result.

Independent

One test should not depend unnecessarily on another test.

Fast

Tests should run quickly enough that developers can run them frequently.

Understandable

A developer should be able to understand what failed.

Deterministic

The same input should normally produce the same expected result.

---

5. The Arrange Act Assert Pattern

A common structure for tests is:

```
Arrange
   ↓
Act
   ↓
Assert
```

Arrange

Prepare the data.

Act

Run the code being tested.

Assert

Check the result.

Example:

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

Test:

```
def test_add():
    # Arrange
    a = 10
    b = 5

    # Act
    result = add(a, b)

    # Assert
    assert result == 15
```

This structure makes tests easier to read.

---

6. Assertions

An assertion checks whether something is true.

Python provides the built-in `assert` statement.

```
assert 2 + 3 == 5
```

This passes because the condition is true.

If the condition is false:

```
assert 2 + 3 == 10
```

Python raises:

```
AssertionError
```

Assertions are useful for simple checks.

```
assert username == "Hafsa"
assert age >= 18
assert len(resources) > 0
```

---

7. Testing a Function

Suppose you have:

```
def multiply(a, b):
    return a * b
```

You can test it with:

```
def test_multiply():
    assert multiply(4, 5) == 20
```

Another example:

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

Tests:

```
def test_even_number():
    assert is_even(4) is True


def test_odd_number():
    assert is_even(5) is False
```

Notice that each test checks one behavior.

---

8. Testing Different Inputs

Testing only one example is rarely enough.

Consider:

```
def square(number):
    return number * number
```

Instead of testing only:

```
assert square(5) == 25
```

test different types of input:

```
assert square(0) == 0
assert square(1) == 1
assert square(5) == 25
assert square(-3) == 9
```

This gives you more confidence.

---

9. Happy Path vs Edge Cases

A **happy path** is the normal expected situation.

Example:

```
def get_first(items):
    return items[0]
```

Happy path:

```
get_first(["Python", "JavaScript"])
```

returns:

```
Python
```

But what happens with:

```
[]
```

That is an edge case.

A good test suite considers both normal and unusual inputs.

Common edge cases

```
0
-1
empty string
empty list
None
```

```
very large number
duplicate values
single item
missing data
invalid input
```

---

10. Testing Expected Failures

Testing is not only about successful results.

Sometimes you expect code to raise an exception.

Example:

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

    return a / b
```

A useful test should verify that invalid input produces the expected error.

With pytest:

```
import pytest


def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)
```

The test passes if `ValueError` is raised.

This is important because an application should fail in the intended way.

---

11. What Is Unit Testing?

A **unit test** checks a small, isolated piece of software.

A unit could be:

- a function
- a method
- a small class behavior

Example:

```
def calculate_discount(price, discount):
    return price - (price * discount)
```

A unit test could be:

```
def test_calculate_discount():
    assert calculate_discount(100, 0.20) == 80
```

The goal is to test the behavior of that unit without involving unrelated systems.

---

12. Unit Tests vs Larger Tests

Software can be tested at different levels.

```
Unit Test
   ↓
Small piece of code

Integration Test
   ↓
Multiple components working together

End-to-End Test
   ↓
Complete user workflow
```

Unit test

```
calculate_total()
```

Integration test

```
Order Service
      ↓
Database
```

End-to-end test

```
User opens website
      ↓
Logs in
      ↓
Adds product
      ↓
Checks out
      ↓
Receives confirmation
```

Each level answers a different question.

---

13. The Testing Pyramid

A common testing model is the testing pyramid.

```
            /\
           /  \
          / E2E\
         /------\
        /Integr. \
       /----------\
      / Unit Tests \
     /--------------\
```

The idea is to have:

- many fast unit tests
- fewer integration tests
- a smaller number of expensive end-to-end tests

Why?

Because unit tests are generally faster and easier to maintain.

For example:

```
100 unit tests
  ↓
20 integration tests
  ↓
5 end-to-end tests
```

The exact numbers are not a rule.

The important idea is balancing speed, coverage, and confidence.

---

14. Introducing pytest

Python has several testing options.

One of the most popular third-party testing frameworks is `pytest`.

Install it:

```
pip install pytest
```

Verify:

```
pytest --version
```

---

15. Your First pytest Test

Create:

```
calculator.py
```

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

Create:

```
test_calculator.py
```

```
from calculator import add


def test_add():
    assert add(2, 3) == 5
```

Run:

```
pytest
```

pytest discovers the test and executes it.

A passing result indicates that the assertion succeeded.

---

16. pytest Test Discovery

pytest automatically searches for test files and functions using naming conventions.

Common test file names include:

```
test_calculator.py
calculator_test.py
```

Test functions commonly start with:

```
test_
```

Example:

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

This naming convention allows pytest to discover tests automatically.

---

17. Multiple Tests

Example:

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


def subtract(a, b):
    return a - b
```

Tests:

```
from calculator import add, subtract


def test_add():
    assert add(10, 5) == 15


def test_subtract():
    assert subtract(10, 5) == 5
```

Running:

```
pytest
```

executes both tests.

---

18. Reading Test Results

Suppose pytest reports:

```
2 passed
```

That means both tests succeeded.

If you see:

```
1 failed, 1 passed
```

one test failed while the other passed.

A failure usually includes:

```
test name
expected result
actual result
```

```
file location
line number
```

Read the failure before changing code.

The test output is evidence about what happened.

---

19. Test Names Matter

Compare:

```
def test_1():
    ...
```

with:

```
def test_discount_is_applied_to_order_total():
    ...
```

The second name communicates intent.

A good test name answers:

> What behavior is this test protecting?

Examples:

```
def test_empty_cart_has_zero_total():
    ...


def test_invalid_email_raises_value_error():
    ...


def test_resource_is_removed_from_collection():
    ...
```

---

20. Testing Return Values

Suppose:

```
def get_full_name(first_name, last_name):
    return f"{first_name} {last_name}"
```

Test:

```
def test_get_full_name():
    result = get_full_name("Hafsa", "Waseem")

    assert result == "Hafsa Waseem"
```

You can also write:

```
assert get_full_name("Hafsa", "Waseem") == "Hafsa Waseem"
```

The first version can sometimes be easier to debug because the intermediate result is visible.

---

21. Testing Collections

Suppose:

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

Tests:

```
def test_get_languages():
    languages = get_languages()

    assert "Python" in languages
    assert len(languages) == 3
```

You can test:

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

if exact ordering and contents are part of the contract.

---

22. Testing Dictionaries

Example:

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

Test:

```
def test_get_user():
    user = get_user()
```

```
    assert user["name"] == "Hafsa"
    assert user["role"] == "developer"
```

You can also test keys:

```
assert "name" in user
```

---

23. Testing Boolean Results

Suppose:

```
def is_authenticated(user):
    return user is not None
```

Test:

```
def test_authenticated_user():
    assert is_authenticated({"name": "Hafsa"}) is True


def test_unauthenticated_user():
    assert is_authenticated(None) is False
```

Using:

```
is True
```

or:

```
is False
```

makes the expected Boolean behavior explicit.

---

24. Testing `None`

Functions frequently return `None`.

Example:

```
def find_resource(resources, resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    return None
```

Test:

```
def test_missing_resource_returns_none():
    resources = [
        {"id": 1, "title": "Python"}
    ]

    result = find_resource(resources, 99)

    assert result is None
```

---

25. Testing Boundaries

Boundary values are particularly important.

Suppose:

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

Testing only:

```
assert is_adult(25) is True
```

does not test the boundary.

Better:

```
assert is_adult(17) is False
assert is_adult(18) is True
assert is_adult(19) is True
```

The value `18` is important because the behavior changes there.

---

26. Test Behavior, Not Implementation

Suppose you have:

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

A useful test checks:

```
assert add(2, 3) == 5
```

It should not care about irrelevant implementation details.

If the function is later changed to:

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

the test should still pass.

Tests should protect the behavior users depend on.

---

27. Avoid Overly Fragile Tests

A fragile test breaks whenever an irrelevant implementation detail changes.

For example, testing private internal variables when the public behavior has not changed can create unnecessary maintenance.

Prefer:

```
assert resource.title == "Python"
```

over checking unrelated internal implementation details.

Good tests allow developers to refactor safely.

---

28. Test Isolation

Tests should ideally be independent.

Bad pattern:

```
test_create_user
       ↓
test_update_user
       ↓
test_delete_user
```

where each test depends on the previous test having run.

Instead, each test should create the state it needs.

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

    update_user(user)

    assert user.name == "New Name"
```

This makes tests easier to run individually.

---

29. Avoid Testing External Systems in Every Unit Test

Imagine a function that sends an email.

```
def send_welcome_email(user):
    ...
```

A unit test should not necessarily send a real email every time.

External systems can be:

- slow
- unavailable
- expensive
- unpredictable

Unit tests often replace external dependencies with controlled test doubles.

This leads to concepts such as:

- mocks
- stubs
- fakes

These are useful when you reach more advanced testing.

---

30. Testing Pure Functions

Pure functions are especially easy to test.

A pure function:

- receives input
- produces output
- does not unexpectedly change external state

Example:

```
def calculate_total(price, quantity):
    return price * quantity
```

Test:

```
def test_calculate_total():
    assert calculate_total(10, 3) == 30
```

No database or file system is required.

This is one reason small, focused functions are valuable.

---

31. Testing Functions With Side Effects

Some functions interact with the outside world.

Examples:

```
write to a file
send an email
call an API
save to a database
modify global state
```

These are side effects.

They require more careful test design.

For example:

```
def save_resource(resource):
    database.save(resource)
```

A unit test may replace the database dependency with a test double rather than using the real database.

---

32. Testing User Input

Suppose:

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

    return True
```

Tests:

```
import pytest


def test_valid_username():
    assert validate_username("Hafsa") is True


def test_short_username():
    with pytest.raises(ValueError):
        validate_username("Hi")
```

Now both valid and invalid behavior are protected.

---

33. Testing Multiple Related Cases

Sometimes a function needs many input combinations.

Instead of manually writing many similar tests, pytest supports parameterization.

Example:

```
import pytest


@pytest.mark.parametrize(
    "number, expected",
    [
        (2, True),
        (4, True),
        (5, False),
        (7, False),
    ]
)
def test_is_even(number, expected):
    assert is_even(number) is expected
```

The same test logic runs with multiple inputs.

This reduces repetitive test code.

---

34. Test Coverage

Test coverage measures how much of the code is exercised by tests.

For example:

```
100 lines of code
80 lines executed by tests
```

could represent roughly:

```
80% line coverage
```

Coverage can be useful, but:

> High coverage does not automatically mean high-quality tests.

This code could technically have high coverage:

```
def calculate_discount(price):
    return price * 0
```

A test that simply executes the function is not enough.

The test should verify correct behavior.

---

35. Coverage vs Confidence

Think of coverage as a measurement tool, not a quality score.

```
Coverage
    ↓
How much code was exercised?

Good tests
    ↓
How much important behavior was checked?
```

A smaller set of meaningful tests can sometimes provide more value than a huge number of meaningless assertions.

---

36. When Should You Write Tests?

Tests can be written:

Before implementation

This is common in Test Driven Development.

```
Write test
   ↓
Test fails
   ↓
Write code
   ↓
Test passes
   ↓
Refactor
```

During implementation

Write tests alongside the feature.

After implementation

Add tests for existing behavior and discovered bugs.

All three approaches can be useful depending on the project.

---

37. Regression Testing

A regression happens when previously working behavior breaks after a change.

Example:

```
Version 1
   ↓
Login works

Developer changes authentication
   ↓
Login breaks
```

A regression test protects against this happening again.

Suppose a bug was discovered:

```
def calculate_total(items):
    ...
```

You create a test that reproduces the bug.

After fixing it:

```
pytest
```

should continue passing.

That test now protects the project from the same bug returning.

---

38. Tests as Documentation

Well-written tests can explain how a function is supposed to behave.

Example:

```
def test_empty_cart_has_zero_total():
    cart = []

    assert calculate_total(cart) == 0
```

A developer reading this immediately learns:

> An empty cart should have a total of zero.

Tests can therefore serve as executable documentation.

---

39. Testing a haas.dev Resource Service

Imagine haas.dev has a resource service:

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

A test:

```
def test_filter_resources_by_category():
    resources = [
        {"title": "Python Basics", "category": "Python"},
        {"title": "React Basics", "category": "React"},
        {"title": "Python Testing", "category": "Python"},
    ]

    result = filter_resources(resources, "Python")

    assert len(result) == 2
    assert result[0]["title"] == "Python Basics"
    assert result[1]["title"] == "Python Testing"
```

This protects an important piece of resource filtering behavior.

---

40. A Practical Testing Workflow

Use this workflow when adding a feature:

Step 1: Define expected behavior

Ask:

> What should this feature do?

Step 2: Identify important cases

Think about:

```
Normal input
Empty input
Invalid input
Boundary values
```

```
Missing data
Expected exceptions
```

Step 3: Write tests

Start with the most important behavior.

Step 4: Implement or modify the code

Write the smallest reasonable implementation.

Step 5: Run tests

```
pytest
```

Step 6: Investigate failures

Do not immediately change the test just because it fails.

Ask:

> Is the code wrong, or is my expectation wrong?

Step 7: Fix the root cause

Step 8: Run the entire suite

```
pytest
```

Step 9: Refactor safely

Once tests pass, improve the implementation while keeping behavior intact.

---

41. Common Testing Mistakes

Mistake 1: Testing only the happy path

Bad:

```
assert calculate_discount(100, 20) == 80
```

and nothing else.

Better:

```
normal value
zero
boundary
invalid value
```

when those cases matter.

Mistake 2: Testing implementation details

Tests should focus on behavior.

Mistake 3: Making tests depend on each other

Each test should generally establish its own state.

Mistake 4: Writing huge tests

A test covering ten unrelated behaviors is difficult to understand.

Prefer focused tests.

Mistake 5: Ignoring failures

A failing test is information.

Do not simply delete or weaken the test to make the suite green.

Mistake 6: Testing only for coverage

Coverage numbers cannot replace meaningful assertions.

Mistake 7: Depending heavily on real external services

Unit tests should generally remain fast and predictable.

Mistake 8: Using unclear test names

Avoid:

```
  def test_function():
```

Prefer:

```
  def test_empty_resource_list_returns_empty_list():
```

---

42. Best Practices

Keep tests small

One test should generally focus on one behavior.

Use descriptive names

Make failures understandable.

Test boundaries

Important behavior often changes at boundaries.

Test invalid input

Applications must handle bad input safely.

Keep tests independent

Avoid hidden dependencies between tests.

Keep unit tests fast

Developers should be able to run them frequently.

Test behavior

Avoid unnecessary coupling to implementation details.

Run tests regularly

Do not wait until the project is finished.

Keep the test suite in version control

Tests are part of the project.

Treat tests as production code

Maintain them with the same care as application code.

---

43. Testing Project Structure

A simple Python project can look like:

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

Keeping tests in a dedicated `tests/` directory makes the project easier to navigate.

---

44. Running Specific Tests

Run all tests:

```
pytest
```

Run one file:

```
pytest tests/test_calculator.py
```

Run one test:

```
pytest tests/test_calculator.py::test_add
```

Run with more detailed output:

```
pytest -v
```

The `-v` option means verbose output.

---

45. A Small Complete Example

Application code:

```
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 pytest

from shop import calculate_total


def test_calculate_total():
    assert calculate_total(10, 3) == 30


def test_zero_quantity():
    assert calculate_total(10, 0) == 0
```

```
def test_negative_price():
    with pytest.raises(ValueError):
        calculate_total(-10, 2)


def test_negative_quantity():
    with pytest.raises(ValueError):
        calculate_total(10, -2)
```

This small suite tests:

```
Normal behavior
Zero boundary
Invalid price
Invalid quantity
```

That is much more useful than testing only one successful input.

---

46. Mini Project: Resource Filter Testing

Build a small resource filtering system.

Application:

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

Create tests for:

1. Matching category
2. No matching category
3. Empty resource list
4. Multiple matching resources
5. Different categories
6. Correct result length
7. Correct resource data

Example:

```
def test_filter_resources_returns_matching_resources():
    resources = [
```

```
        {"title": "Python", "category": "programming"},
        {"title": "CSS", "category": "web"},
        {"title": "Testing", "category": "programming"},
    ]

    result = filter_resources(resources, "programming")

    assert len(result) == 2
    assert result[0]["title"] == "Python"
    assert result[1]["title"] == "Testing"
```

Then run:

```
pytest
```

---

47. 5 Minute Challenge

Create:

```
def calculate_average(numbers):
    ...
```

Write tests for:

```
[10, 20, 30] → 20
[5] → 5
[0, 0, 0] → 0
[]
```

Decide what should happen for an empty list.

Should the function:

```
return 0
```

or:

```
raise ValueError
```

Choose a behavior and write a test that documents that decision.

---

48. 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"]
```

Also consider:

```
[]
["Python"]
```

Exercise 3

Create:

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

Require:

```
at least 8 characters
```

Write tests for:

```
valid password
7-character password
8-character password
empty password
```

Exercise 4

Create:

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

Define your own rules:

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

Write tests for values around the boundary:

```
49
50
51
```

Exercise 5

Create a small resource service with:

```
add_resource()
remove_resource()
find_resource()
```

Write tests for all three operations.

Include:

```
normal cases
missing resources
empty collections
duplicate IDs
```

where applicable.

---

49. Mini Quiz

1. What is the main purpose of testing?

A. Make code longer

B. Check expected behavior

C. Replace debugging

D. Remove all bugs automatically

Answer: B

---

2. What does an assertion do?

A. Installs a package

B. Checks a condition

C. Creates a class

D. Starts a server

Answer: B

---

3. Which command commonly runs pytest?

A. `python test`

B. `run pytest`

C. `pytest`

D. `start-tests`

Answer: C

---

4. What should a good unit test generally focus on?

A. An entire application

B. One small behavior

C. Every possible feature

D. The user interface only

Answer: B

---

5. Why test edge cases?

A. They make code look advanced

B. They often expose incorrect behavior

C. They replace normal tests

D. They remove the need for debugging

Answer: B

---

50. Interview Questions

Beginner

1. What is software testing?
2. Why is testing important?
3. What is a unit test?
4. What is an assertion?

5. What is pytest?
6. How do you run pytest?
7. What is the Arrange Act Assert pattern?
8. What is an edge case?
9. What is regression testing?
10. Why should tests have descriptive names?

Intermediate

11. What is the difference between unit, integration, and end-to-end testing?
12. What is the testing pyramid?
13. Why should unit tests be isolated?
14. What makes a test deterministic?
15. What is test coverage?
16. Why doesn't 100% coverage guarantee bug-free software?
17. What is parameterized testing?
18. Why should tests avoid unnecessary implementation details?
19. When should you test exceptions?
20. What are mocks, stubs, and fakes?

Practical

21. How would you test a function that receives invalid input?
22. How would you test a function with multiple boundary values?
23. How would you test code that interacts with a database?
24. How would you investigate a failing test?
25. How can tests help during refactoring?

---

51. Testing Mental Model

When writing tests, think:

```
What should happen?
        ↓
What inputs matter?
        ↓
What should the output be?
        ↓
What could go wrong?
        ↓
```

```
What are the boundaries?
        ↓
What should happen for invalid input?
        ↓
Write focused tests
        ↓
Run tests
        ↓
Fix failures
        ↓
Refactor safely
```

The goal is not:

> "How can I test every line?"

The better question is:

> "What behavior must remain correct?"

---

52. Testing Cheat Sheet

Install pytest

```
pip install pytest
```

Run tests

```
pytest
```

Verbose output

```
pytest -v
```

Run one file

```
pytest tests/test_file.py
```

Run one test

```
pytest tests/test_file.py::test_name
```

Basic assertion

```
assert result == expected
```

Boolean

```
assert result is True
assert result is False
```

None

```
assert result is None
```

Collection

```
assert item in items
assert len(items) == 3
```

Expected exception

```
with pytest.raises(ValueError):
    function()
```

Parameterized test

```
@pytest.mark.parametrize(
    "input_value, expected",
    [
        (1, True),
        (2, False),
    ]
)
def test_something(input_value, expected):
    assert function(input_value) is expected
```

---

53. Key Takeaways

- Testing checks whether software behaves as expected.
- Testing and debugging are different activities.
- Assertions verify expected conditions.
- Unit tests focus on small pieces of behavior.
- Good tests are focused, readable, repeatable, and independent.
- Test both normal behavior and important edge cases.
- Invalid inputs and expected exceptions should also be tested.
- pytest is a popular framework for Python testing.
- Test names should describe behavior.
- Tests should focus on behavior rather than unnecessary implementation details.

- Test coverage is useful but does not measure test quality by itself.
- Regression tests protect previously fixed behavior.
- Tests make refactoring safer.
- A strong test suite becomes executable documentation for the project.

The core idea is simple:

```
Code tells the computer what to do.

Tests tell you whether it is still doing what you intended.
```

Visit haas.dev for more resources and guides.

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

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