**Writing tests with GitHub Copilot**

GitHub Copilot can assist you in developing tests quickly and improving productivity. In this article, we’ll demonstrate how you can use Copilot to write both unit and integration tests. While Copilot performs well when generating tests for basic functions, complex scenarios require more detailed prompts and strategies. A walk through practical examples of using Copilot to break down tasks and verify code correctness.

**Writing unit tests with Copilot Chat**

We’ll explore how to use GitHub Copilot Chat to generate unit tests for a Python class. This example demonstrates how you can use Copilot to create unit tests for a class like BankAccount. We will show you how to prompt Copilot to generate tests, execute them, and verify the results.

**Example class: BankAccount**

Let’s start with a class BankAccount that contains methods for depositing, withdrawing, and getting the balance of an account. Create a new file bank\_account.py in a GitHub repository and add the following BankAccount class in Python.

class BankAccount:

def \_\_init\_\_(self, initial\_balance=0):

if initial\_balance < 0:

raise ValueError("Initial balance cannot be negative.")

self.balance = initial\_balance

def deposit(self, amount):

if amount <= 0:

raise ValueError("Deposit amount must be positive.")

self.balance += amount

def withdraw(self, amount):

if amount <= 0:

raise ValueError("Withdrawal amount must be positive.")

if amount > self.balance:

raise ValueError("Cannot withdraw more than the current balance.")

self.balance -= amount

def get\_balance(self):

return self.balance

**Generating unit tests with Copilot Chat**

You can prompt Copilot with a request to write tests that cover a range of scenarios. Be specific with your test requirements to get the best results. The following is an example prompt you can use to generate unit tests for the BankAccount class.

**Prompt:** *Develop a comprehensive suite of unit tests for the BankAccount() function in Python. Write multiple test methods that cover a wide range of scenarios, including edge cases, exception handling, and data validation.*

This prompt does a few important things:

* **Scope:** It explicitly asks for a comprehensive suite of unit tests, so that Copilot generates more than just basic tests, covering a variety of potential use cases.
* **Scenarios:** The prompt specifies that the tests should include:
  + **Edge cases:** Tests for boundary conditions (e.g., negative values for deposits or withdrawals, withdrawing more than the available balance).
  + **Exception handling:** Tests for errors that should be raised when certain invalid actions occur (e.g., withdrawing more than the available balance, depositing negative amounts).
  + **Data validation:** Ensures that inputs are correctly validated and handled by the methods.

Copilot will give you a response like the following.

import unittest

from bank\_account import BankAccount

class TestBankAccount(unittest.TestCase):

def setUp(self):

self.account = BankAccount()

def test\_initial\_balance(self):

self.assertEqual(self.account.get\_balance(), 0)

def test\_deposit\_positive\_amount(self):

self.account.deposit(100)

self.assertEqual(self.account.get\_balance(), 100)

def test\_withdraw\_within\_balance(self):

self.account.deposit(100)

self.account.withdraw(50)

self.assertEqual(self.account.get\_balance(), 50)

def test\_deposit\_negative\_amount\_raises\_error(self):

with self.assertRaises(ValueError):

self.account.deposit(-100)

def test\_withdraw\_negative\_amount\_raises\_error(self):

with self.assertRaises(ValueError):

self.account.withdraw(-50)

def test\_withdraw\_more\_than\_balance\_raises\_error(self):

self.account.deposit(100)

with self.assertRaises(ValueError):

self.account.withdraw(200)

def test\_initial\_balance\_negative\_raises\_error(self):

with self.assertRaises(ValueError):

BankAccount(-100)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Running the tests cases**

The test suite generated by Copilot includes a test case for each scenario specified in the prompt. Each function in the BankAccount class is tested for a range of conditions, including valid operations like deposits and withdrawals, as well as edge cases such as negative amounts and attempts to withdraw more than the available balance.

Once Copilot has generated the test suite to your satisfaction, add the code to a new file test\_bank\_account.py. You can ask it how to run the tests.

**Prompt:** *"How do I run these unit tests in Python using the unittest framework?"*

Copilot will give you the following bash command.

python -m unittest test\_bank\_account.py

After running the tests, you will see the output in your terminal or IDE. If all tests pass, you can be confident that your BankAccount class is working as expected.

**Slash command**

Additionally, you can prompt Copilot to write a full suite of unit tests with the /tests slash command. Ensure that you have the file open on the current tab of your IDE and Copilot will generate unit tests for that file. The tests that Copilot generates may not cover all scenarios, so you should always review the generated code and add any additional tests that may be necessary.

*Tip: If you ask Copilot to write tests for a code file that is not already covered by unit tests, you can provide Copilot with useful context by opening one or more existing test files in adjacent tabs in your editor. Copilot will be able to see the testing framework you use and will be more likely to write a test that is consistent with your existing tests.*

Copilot will generate a unit test suite such as the following.

import unittest

from bank\_account import BankAccount

class TestBankAccount(unittest.TestCase):

def setUp(self):

self.account = BankAccount()

def test\_initial\_balance(self):

self.assertEqual(self.account.get\_balance(), 0)

**Writing integration tests with Copilot**

Integration tests are essential for ensuring that the various components of your system work correctly when combined. In this section, we’ll extend our BankAccount class to include interactions with an external service NotificationSystem and use mocks to test the system’s behavior without needing real connections. The goal of the integration tests is to verify the interaction between the BankAccount class and the NotificationSystem services, ensuring that they work together correctly.

**Example class: BankAccount with notification services**

Let's update the BankAccount class to include interactions with an external service such as a NotificationSystem that sends notifications to users. NotificationSystem represents the integration that would need to be tested.

Update the BankAccount class in the bank\_account.py file with the following code snippet.

class BankAccount:

def \_\_init\_\_(self, initial\_balance=0, notification\_system=None):

if initial\_balance < 0:

raise ValueError("Initial balance cannot be negative.")

self.balance = initial\_balance

self.notification\_system = notification\_system

def deposit(self, amount):

if amount <= 0:

raise ValueError("Deposit amount must be positive.")

self.balance += amount

if self.notification\_system:

self.notification\_system.notify(f"Deposited {amount}, new balance: {self.balance}")

def withdraw(self, amount):

if amount <= 0:

raise ValueError("Withdrawal amount must be positive.")

if amount > self.balance:

raise ValueError("Cannot withdraw more than the current balance.")

self.balance -= amount

if self.notification\_system:

self.notification\_system.notify(f"Withdrew {amount}, new balance: {self.balance}")

def get\_balance(self):

return self.balance

Here we'll break down our request for Copilot to write integration tests for the BankAccount class into smaller, more manageable pieces. This will help Copilot generate more accurate and relevant tests.

**Prompt:** *"Write integration tests for the deposit function in the BankAccount class. Use mocks to simulate the NotificationSystem and verify that it is called correctly after a deposit."*

This prompt does a few important things:

* **Scope:** It specifies integration tests, focusing on the interaction between the deposit function and the NotificationSystem, rather than just unit tests.
* **Mocks:** It explicitly asks for the use of mocks to simulate the NotificationSystem, ensuring that the interaction with external systems is tested without relying on their actual implementation.
* **Verification:** The prompt emphasizes verifying that the NotificationSystem is called correctly after a deposit, ensuring that the integration between the components works as expected.
* **Specificity:** The prompt clearly states the method (deposit) and the class (BankAccount) to be tested.

Tip

If Copilot is producing invalid tests, provide examples of inputs and outputs for the function you want to test. This will help Copilot evaluate the expected behaviour of the function.

Copilot will generate a test suite like the following.

import unittest

from unittest.mock import Mock

from bank\_account import BankAccount

class TestBankAccountIntegration(unittest.TestCase):

def setUp(self):

self.notification\_system = Mock()

def test\_deposit\_with\_notification(self):

account = BankAccount(initial\_balance=100, notification\_system=self.notification\_system)

account.deposit(50)

self.assertEqual(account.get\_balance(), 150)

self.notification\_system.notify.assert\_called\_once\_with("Deposited 50, new balance: 150")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Add the generated code to a new file test\_bank\_account\_integration.py.

**Improving on the test cases**

The prompt above generated a single test case that verifies the NotificationSystem is called when a valid deposit is made. However, it doesn't cover cases where an error is raised during the deposit. In those scenarios, the NotificationSystem should not be called. We need to add a test case that handles invalid deposits and ensure the notification system is not triggered.

**Prompt:** *"Add a test case for invalid deposit amounts to verify the function raises the correct exceptions and that the NotificationService is not called."*

Copilot will generate a test case such as the following.

def test\_deposit\_negative\_amount\_raises\_error(self):

account = BankAccount(initial\_balance=100, notification\_system=self.notification\_system)

with self.assertRaises(ValueError):

account.deposit(0)

self.notification\_system.notify.assert\_not\_called()

**Asking about areas of improvement**

Now that we've written test cases to validate the integration functionality for deposits, this is a great opportunity to search for improvements within the test suite. While the current tests are functional, we can prompt Copilot to evalulate code coverage and suggest areas of improvement.

**Prompt:** *"What additional tests should be included to ensure full coverage for the integration between the BankAccount class and the NotificationSystem?"*

Prompting Copilot with this question can help you identify missing test cases that may have been overlooked. In this situation, while we tested valid and invalid deposits, we haven't yet covered the withdrawal functionality.

Copilot will generate an updated test suite such as the following.

Click to expand the full generated code example

Once Copilot has generated the test suite to your satisfaction, run the tests with command below to verify the results.

python -m unittest test\_bank\_account\_integration.py

**Using Copilot Spaces to improve test suggestions**

Copilot Spaces is a feature that allows you to organize and share task-specific context with Copilot. This can help improve the relevance of the suggestions you receive. By providing Copilot with more context about your project, you can get better test suggestions.

For example, you could create a space that includes:

* The module you’re testing (like payments.js)
* The current test suite (like payments.test.js)
* A test coverage report or notes about what's missing

In the space, you can ask Copilot questions like:

What test cases are missing in payments.test.js based on the logic in payments.js?

Or:

Write a unit test for the refund logic in refund.js, following the structure in the existing test suite.