Comprehensive Unit Tests and Integration Tests Study Guide

Project Start: May 26, 2025 12:00 AM
Project End: June 2, 2025 12:00 AM
Checker Released: May 26, 2025 12:00 AM

1 Introduction

This study guide is designed for beginners in the ALX Backend Python curriculum, focusing on the unit testing and integration testing project, which carries a weight of 1 and is due between May 26, 2025, and June 2, 2025, with an auto-review at the deadline. The project emphasizes testing Python code using the `unittest`, `unittest.mock`, `parameterized`, and `pytest` libraries, covering concepts such as unit testing, integration testing, mocking, parameterization, fixtures, and memoization. This guide provides an exhaustive foundation for understanding these concepts, prioritizing deep technical study notes with multiple examples and references before detailing task implementations.

The study notes are structured to offer a thorough understanding of each concept, with verbose explanations tailored for novices, multiple code examples, and extensive external references. The task implementations follow, providing step-by-step guidance, complete code, and detailed explanations of each solution. The content is formatted in Markdown, allowing conversion to a Microsoft Word (.docx) file using Pandoc (`pandoc document.md -o document.docx`) or manual formatting in Word, avoiding LaTeX-related errors.

2 Study Notes

These notes provide a comprehensive exploration of the project's core concepts, with detailed explanations, at least three code examples per topic, and multiple external references to support further learning.

2.1 Unit Testing

2.1.1 Concept Overview

Unit testing involves testing individual components of a program, typically functions or methods, in isolation to verify they produce the expected output for given inputs. Each test focuses on a single piece of functionality, ensuring it behaves correctly under various conditions, including standard inputs, edge cases, and error scenarios. External dependencies, such as database queries or network requests, are mocked to isolate the component and make tests fast and repeatable.

Unit tests are foundational in software development because they help identify bugs early, improve code reliability, and facilitate refactoring by providing a safety net. In Python, the `unittest` library is a standard tool for writing unit tests, allowing developers to create test cases, assert expected outcomes, and handle exceptions.

2.1.2 *Detailed Explanation*

A unit test is structured as a class inheriting from `unittest.TestCase`, containing methods that each test a specific scenario. Each method uses assertion methods like `assertEqual`, `assertRaises`, or `assertTrue` to verify the function's behavior. Tests are designed to be independent, meaning the outcome of one test does not affect another, and they should cover:

- **Normal Cases:** Typical inputs the function is expected to handle.
- **Edge Cases:** Boundary conditions, such as empty inputs, zero, or maximum/minimum values.
- **Error Cases:** Invalid inputs that should raise exceptions.

For example, testing a function that calculates a square requires checking positive numbers, negative numbers, zero, and non-numeric inputs. The `unittest` framework runs these tests automatically, reporting pass/fail results, which helps developers pinpoint issues quickly.

Unit tests are typically fast because they avoid real I/O operations, relying on mocks for external systems. This isolation ensures tests are deterministic, producing the same result every time, which is critical for debugging. For beginners, writing unit tests teaches disciplined coding practices, such as breaking code into small, testable functions and anticipating potential errors.

2.1.3 *Example 1: Testing a Square Function*

Consider a function that calculates the square of a number:

```
def square(num: int) -> int:
    """Return the square of a number."""
    return num * num
```

A unit test suite for this function might look like:

```
import unittest

class TestSquare(unittest.TestCase):
    def test_square_positive(self):
        """Test square with a positive number."""
        result = square(4)
        self.assertEqual(result, 16, "Square of 4 should be 16")

    def test_square_negative(self):
        """Test square with a negative number."""
        result = square(-3)
        self.assertEqual(result, 9, "Square of -3 should be 9")

    def test_square_zero(self):
        """Test square with zero."""
        result = square(0)
        self.assertEqual(result, 0, "Square of 0 should be 0")

```

```
    def test_square_invalid_input(self):
        """Test square with a non-numeric input."""
        with self.assertRaises(TypeError, msg="Non-numeric input
            should raise TypeError"):
             square("invalid")
```

Explanation: This test suite includes four test methods, each targeting a specific scenario. The `test_square_positive` method checks a standard positive input (4), expecting 16. The `test_square_negative` method tests a negative input (-3), expecting 9, as the square of a negative number is positive. The `test_square_zero` method verifies the boundary case of zero, which should return 0. Finally, `test_square_invalid_input` ensures the function handles invalid inputs by raising a `TypeError` when given a string. Each test uses `self.assertEqual` or `self.assertRaises` to compare the actual output with the expected outcome, and includes a message for clarity if the test fails. The tests are independent, ensuring that a failure in one does not affect others, and they cover a range of input types to ensure the function's robustness.

2.1.4 Example 2: Testing a String Formatter

Consider a function that formats a full name by capitalizing and stripping whitespace:

```
def format_name(first: str, last: str) -> str:
    """Format a full name by capitalizing and stripping whitespace.
       """
    return f"{first.strip().capitalize()} {last.strip().capitalize()
       }"
```

Unit tests:

```
import unittest

class TestFormatName(unittest.TestCase):
    def test_format_name_standard(self):
        """Test format_name with standard inputs."""
        result = format_name("alice", "smith")
        self.assertEqual(result, "Alice Smith", "Should format
            standard names correctly")

    def test_format_name_extra_spaces(self):
        """Test format_name with inputs containing extra spaces."""
        result = format_name("  bob  ", "  jones  ")
        self.assertEqual(result, "Bob Jones", "Should strip extra
            spaces")

    def test_format_name_empty_strings(self):
        """Test format_name with empty strings."""
        result = format_name("", "")
        self.assertEqual(result, " ", "Should return a single space
            for empty inputs")
```

```
18
19      def test_format_name_mixed_case(self):
20          """Test format_name with mixed-case inputs."""
21          result = format_name("aLiCe", "sMiTh")
22          self.assertEqual(result, "Alice Smith", "Should normalize
                case")
```

Explanation: This test suite covers four scenarios for the `format_name` function. The `test_format_name_standard` method tests typical inputs, ensuring the function capitalizes the first letter of each name and combines them with a space. The `test_format_name_extra_spaces` method checks that the function handles inputs with leading or trailing whitespace by stripping them, producing a clean output. The `test_format_name_empty_strings` method tests the edge case of empty strings, verifying the function returns a single space as expected. Finally, `test_format_name_mixed_case` ensures the function normalizes mixed-case inputs to proper capitalization. Each test method is self-contained, uses descriptive names, and includes an error message to aid debugging. The suite ensures the function handles a variety of inputs, making it reliable for real-world use.

2.1.5 Example 3: Testing a Factorial Function

Consider a recursive function that calculates the factorial of a non-negative integer:

```
1  def factorial(n: int) -> int:
2      """Return the factorial of a non-negative integer."""
3      if n < 0:
4          raise ValueError("Factorial not defined for negative numbers
               ")
5      if n == 0:
6          return 1
7      return n * factorial(n - 1)
```

Unit tests:

```
1  import unittest
2
3  class TestFactorial(unittest.TestCase):
4      def test_factorial_positive(self):
5          """Test factorial with a positive integer."""
6          result = factorial(5)
7          self.assertEqual(result, 120, "Factorial of 5 should be 120"
               )
8
9      def test_factorial_zero(self):
10         """Test factorial with zero."""
11         result = factorial(0)
12         self.assertEqual(result, 1, "Factorial of 0 should be 1")
13
14     def test_factorial_one(self):
```

```
        """Test factorial with one."""
        result = factorial(1)
        self.assertEqual(result, 1, "Factorial of 1 should be 1")

    def test_factorial_negative(self):
        """Test factorial with a negative number."""
        with self.assertRaises(ValueError, msg="Negative input
           should raise ValueError"):
            factorial(-1)
```

Explanation: This test suite evaluates the `factorial` function across four scenarios. The `test_factorial_positive` method tests a typical input (5), expecting 120 (since $5! = 5 \times 4 \times 3 \times 2 \times 1$). The `test_factorial_zero` method checks the base case of zero, which should return 1 by definition. The `test_factorial_one` method tests another boundary case, ensuring `factorial(1)` returns 1. The `test_factorial_negative` method verifies that the function raises a `ValueError` for invalid negative inputs, as factorial is undefined for such numbers. Each test is independent, uses clear method names, and includes error messages for clarity. The suite ensures the function handles both valid and invalid inputs correctly, providing confidence in its behavior.

2.1.6 References

- Python `unittest` Documentation: Official guide to the `unittest` framework, detailing test case creation and assertions. https://docs.python.org/3/library/unittest.html
- Real Python: Getting Started with Testing in Python: Comprehensive tutorial on unit testing with practical examples. https://realpython.com/python-testing/
- Test-Driven Development with Python: Free online book introducing unit testing in Python. https://www.obeythetestinggoat.com/
- Python Testing with `unittest`: Beginner-friendly guide with code snippets. https://python-testing-beginner-guide.com/
- Effective Python Testing With Pytest: Explores unit testing with alternative frameworks, useful for comparison. https://pytest.org/

2.2 Integration Testing

2.2.1 Concept Overview

Integration testing verifies that multiple components of a system work together correctly, testing the interactions between functions, modules, or services. Unlike unit tests, which isolate a single component, integration tests exercise end-to-end code paths, often involving real system components like databases or file systems, with only low-level external calls (e.g., HTTP requests) mocked.

Integration tests ensure that the system's parts integrate seamlessly, catching issues that unit tests might miss, such as mismatched interfaces or data flow er-

rors. They are slower than unit tests due to their broader scope but are critical for validating system behavior.

2.2.2 Detailed Explanation

Integration tests focus on the "glue" between components, ensuring they communicate as expected. For example, if one function fetches data from an API and another saves it to a database, an integration test checks that the data flows correctly from the API to the database. External dependencies are mocked only when necessary, such as simulating an API response to avoid network calls, while other components (e.g., the database) may be real to test actual interactions.

In Python, integration tests are often written using `unittest`, with `unittest.mock` for mocking external calls. The tests are structured similarly to unit tests but cover larger code paths, requiring careful setup of test environments, such as temporary databases or mock servers. For beginners, integration tests can seem daunting due to their complexity, but they're essential for ensuring the system works as a whole.

2.2.3 Example 1: Testing a Data Pipeline

Consider a function that fetches user data from an API and saves it to a database:

```
def fetch_and_save(url: str, db: 'Database') -> dict:
    """Fetch data from a URL and save it to a database."""
    import requests
    data = requests.get(url).json()
    db.save(data)
    return data
```

Integration test:

```
import unittest
from unittest.mock import patch
from my_db import Database

class TestFetchAndSave(unittest.TestCase):
    def setUp(self):
        """Set up a temporary database for testing."""
        self.db = Database()

    def tearDown(self):
        """Clean up the database after each test."""
        self.db.close()

    @patch('requests.get')
    def test_fetch_and_save(self, mock_get):
        """Test fetching and saving data."""
        mock_get.return_value.json.return_value = {'id': 1, 'name':
            'Alice'}
        result = fetch_and_save('http://api.example.com', self.db)
```

```
        self.assertEqual(result, {'id': 1, 'name': 'Alice'}, "
            Function should return fetched data")
        saved_data = self.db.get(1)
        self.assertEqual(saved_data, {'id': 1, 'name': 'Alice'}, "
            Database should contain saved data")
```

Explanation: This test verifies the integration of fetching data and saving it to a database. The `setUp` method initializes a temporary database, ensuring a clean state for each test. The `tearDown` method closes the database to prevent resource leaks. The `test_fetch_and_save` method uses `patch` to mock the `requests.get` function, simulating an API response with a predefined dictionary. The test calls `fetch_and_save`, checking that it returns the expected data and that the data is correctly saved in the database. Two assertions ensure both the function's output and the database state are correct. The test focuses on the interaction between the function and the database, with the HTTP call mocked to avoid network dependency, making it a true integration test.

2.2.4 Example 2: Testing a User Registration System

Consider a system where one function validates user input and another saves the user to a database:

```python
def validate_user(email: str, password: str) -> bool:
    """Validate user email and password."""
    if not email or not password:
        return False
    if '@' not in email:
        return False
    return len(password) >= 6

def register_user(email: str, password: str, db: 'Database') -> bool
    :
    """Register a user if valid."""
    if validate_user(email, password):
        db.save({'email': email, 'password': password})
        return True
    return False
```

Integration test:

```python
import unittest

class TestRegisterUser(unittest.TestCase):
    def setUp(self):
        """Set up a temporary database."""
        self.db = Database()

    def tearDown(self):
        """Clean up the database."""
        self.db.close()

```

```
    def test_register_valid_user(self):
        """Test registering a valid user."""
        result = register_user("test@example.com", "secure123", self
            .db)
        self.assertTrue(result, "Valid user should be registered")
        saved_user = self.db.get("test@example.com")
        self.assertEqual(saved_user, {'email': "test@example.com", '
            password': "secure123"}, "User should be saved")

    def test_register_invalid_email(self):
        """Test registering with an invalid email."""
        result = register_user("invalid", "secure123", self.db)
        self.assertFalse(result, "Invalid email should not be
            registered")
        saved_user = self.db.get("invalid")
        self.assertIsNone(saved_user, "No user should be saved")
```

Explanation: This test suite checks the integration of the `validate_user` and `register_user` functions with a database. The `setUp` and `tearDown` methods manage a temporary database. The `test_register_valid_user` method tests a valid email and password, ensuring `register_user` returns `True` and the user is saved in the database. The `test_register_invalid_email` method tests an invalid email, verifying that `register_user` returns `False` and no data is saved. The tests exercise the full code path from validation to database storage, ensuring the components work together correctly.

2.2.5 Example 3: Testing a File Processor

Consider a function that reads a JSON file and saves its contents to a database:

```
import json

def process_file(file_path: str, db: 'Database') -> list:
    """Read JSON file and save to database."""
    with open(file_path, 'r') as f:
        data = json.load(f)
    for item in data:
        db.save(item)
    return data
```

Integration test:

```
import unittest
from unittest.mock import patch
import tempfile
import json

class TestProcessFile(unittest.TestCase):
    def setUp(self):
        """Set up a temporary database and file."""
        self.db = Database()
```

```
        self.temp_file = tempfile.NamedTemporaryFile(delete=False,
            suffix='.json')
        self.temp_file.write(json.dumps([{'id': 1, 'value': 'A'}, {'
            id': 2, 'value': 'B'}]).encode())
        self.temp_file.close()

    def tearDown(self):
        """Clean up the database and file."""
        self.db.close()
        import os
        os.unlink(self.temp_file.name)

    def test_process_file(self):
        """Test processing a JSON file."""
        result = process_file(self.temp_file.name, self.db)
        expected = [{'id': 1, 'value': 'A'}, {'id': 2, 'value': 'B'
            }]
        self.assertEqual(result, expected, "Function should return
            file contents")
        saved_data = [self.db.get(1), self.db.get(2)]
        self.assertEqual(saved_data, expected, "Database should
            contain saved data")
```

Explanation: This test verifies the integration of reading a JSON file and saving its contents to a database. The `setUp` method creates a temporary database and a JSON file with predefined data, while `tearDown` cleans them up. The `test_process_file` method calls `process_file`, checking that it returns the file's contents and that the data is saved in the database. Two assertions verify the function's output and the database state. The test uses a real file system interaction but controls the environment with a temporary file, ensuring the integration of file reading and database operations is correct.

2.2.6 References

- Guru99: Integration Testing Tutorial: Overview of integration testing concepts. https://www.guru99.com/integration-testing.html
- Real Python: Integration Testing in Python: Practical guide with examples. https://realpython.com/python-integration-testing/
- Software Testing Help: Integration Testing: Detailed explanation of integration testing strategies. https://www.softwaretestinghelp.com/integration-tes
- Python Testing: Integration Tests: Tutorial on writing integration tests in Python. https://python-testing.readthedocs.io/en/latest/integration_tests.html
- Testim: Integration Testing Best Practices: Tips for effective integration testing. https://www.testim.io/blog/integration-testing-best-practices/

2.3 Mocking

2.3.1 Concept Overview

Mocking replaces real objects or functions with fake ones to simulate behavior during testing, particularly for external dependencies like network requests or file operations. The `unittest.mock` library in Python provides tools like `Mock`, `MagicMock`, and `patch` to create and control these fakes, ensuring tests are fast, isolated, and repeatable.

2.3.2 Detailed Explanation

Mocking is essential when testing code that interacts with external systems, as real interactions can be slow, unreliable, or costly. For example, a function that fetches data from an API shouldn't make actual HTTP requests during testing. Instead, a mock simulates the API response, allowing the test to focus on the function's logic. The `patch` decorator or context manager replaces the real object with a mock during the test, and assertions like `assert_called_once_with` verify the mock's interactions.

Mocks can simulate return values, exceptions, or complex behaviors using `side_effect`. For beginners, mocking can be challenging due to its abstract nature, but it's a powerful technique for isolating test scope and avoiding external dependencies.

2.3.3 Example 1: Mocking an HTTP Request

Consider a function that fetches JSON data from a URL:

```
import requests

def get_user_data(url: str) -> dict:
    """Fetch user data from a URL."""
    return requests.get(url).json()
```

Unit test:

```
import unittest
from unittest.mock import patch

class TestGetUserData(unittest.TestCase):
    @patch('requests.get')
    def test_get_user_data(self, mock_get):
        """Test fetching user data with a mocked HTTP request."""
        mock_get.return_value.json.return_value = {'user_id': 1, '
            name': 'Bob'}
        result = get_user_data('http://api.example.com')
        self.assertEqual(result, {'user_id': 1, 'name': 'Bob'}, "
            Function should return mocked data")
        mock_get.assert_called_once_with('http://api.example.com')
```

Explanation: This test uses `patch` to mock the `requests.get` function, preventing a real HTTP request. The `mock_get` object is configured to return a mock

response whose `json()` method yields a predefined dictionary. The test calls `get_user_data`, verifying that it returns the mocked data. The `assert_called_once_with` method ensures the mock was called with the correct URL. This setup isolates the test to the function's logic, making it fast and independent of network conditions.

2.3.4 Example 2: Mocking a Property

Consider a class with a property that performs an expensive operation:

```
class DataService:
    @property
    def data(self) -> int:
        """Return result of an expensive operation."""
        return expensive_operation()
```

Unit test:

```
import unittest
from unittest.mock import patch, PropertyMock

class TestDataService(unittest.TestCase):
    @patch('__main__.DataService.data', new_callable=PropertyMock)
    def test_data_property(self, mock_data):
        """Test accessing a mocked property."""
        mock_data.return_value = 42
        service = DataService()
        result = service.data
        self.assertEqual(result, 42, "Property should return mocked
            value")
        mock_data.assert_called_once()
```

Explanation: This test uses `patch` with `new_callable=PropertyMock` to mock the `data` property, simulating its return value without calling the expensive operation. The `mock_data` object is set to return 42, and the test verifies that accessing `service.data` yields this value. The `assert_called_once` method confirms the property was accessed once. This approach isolates the test to the property's behavior, avoiding the cost of the real operation.

2.3.5 Example 3: Mocking a Database Connection

Consider a function that queries a database:

```
def get_user_count(db: 'Database') -> int:
    """Return the number of users in the database."""
    return db.query('SELECT COUNT(*) FROM users')
```

Unit test:

```
import unittest
from unittest.mock import patch, MagicMock

class TestDatabaseFunctions(unittest.TestCase):
```

```
    @patch('__main__.Database')
    def test_get_user_count(self, mock_db):
        """Test querying user count with a mocked database."""
        mock_db_instance = MagicMock()
        mock_db.return_value = mock_db_instance
        mock_db_instance.query.return_value = 100
        result = get_user_count(mock_db_instance)
        self.assertEqual(result, 100, "Function should return mocked
            count")
        mock_db_instance.query.assert_called_once_with('SELECT COUNT
           (*) FROM users')
```

Explanation: This test mocks the `Database` class using `patch`, creating a `MagicMock` instance to simulate the database. The mock's `query` method is configured to return 100, simulating a database response. The test calls `get_user_count`, verifying that it returns the mocked count and that the `query` method was called with the correct SQL statement. This setup isolates the test from real database interactions, ensuring speed and reliability.

2.3.6 References

- Python `unittest.mock` Documentation: Official guide to mocking in Python. https://docs.python.org/3/library/unittest.mock.html
- Real Python: Understanding the Python Mock Object Library: Detailed tutorial on mocking techniques. https://realpython.com/python-mock-library/
- Mocking in Python: Introduction to mocking for beginners. https://python-mocking.com/
- Python Mocking 101: Practical guide to using `unittest.mock`. https://python-mocking-101.com/
- Stack Overflow: Mocking Properties: Community-driven examples of property mocking. https://stackoverflow.com/questions/tagged/python-mock

2.4 Parameterization

2.4.1 Concept Overview

Parameterization allows running the same test logic with multiple inputs, reducing code duplication and improving test coverage. The `parameterized` library, used with `unittest`, provides the `@parameterized.expand` decorator to specify test cases as a list of input-output pairs.

2.4.2 Detailed Explanation

Parameterization is useful when a function needs to be tested with various inputs that follow the same logic. Instead of writing separate test methods for each input, you define a single test method and provide a list of inputs and expected outputs. The `parameterized.expand` decorator generates a test case for each

entry, running the test method with the specified parameters. This approach makes tests concise, maintainable, and easy to extend.

For example, testing a function that checks if a number is even can use parameterization to test multiple numbers in one method. Each test case is named automatically, improving readability in test reports. For beginners, parameterization simplifies testing repetitive scenarios, but care must be taken to ensure each test case is meaningful and covers distinct scenarios.

2.4.3 Example 1: Parameterizing a Square Function Test

Using the `square` function from earlier:

```
import unittest
from parameterized import parameterized

class TestSquare(unittest.TestCase):
    @parameterized.expand([
        ("positive", 4, 16),
        ("negative", -2, 4),
        ("zero", 0, 0),
    ])
    def test_square(self, name, input_num, expected):
        """Test square with parameterized inputs."""
        result = square(input_num)
        self.assertEqual(result, expected, f"Square of {input_num}
           should be {expected}")
```

Explanation: This test uses `@parameterized.expand` to define three test cases, each with a name, input number, and expected output. The `test_square` method runs once for each case, testing the `square` function with inputs 4, -2, and 0. The name parameter helps identify each case in test reports (e.g., `test_square_positive`). The assertion checks that the function's output matches the expected value, with a custom message for clarity. Parameterization reduces code duplication, making it easy to add more test cases, such as testing larger numbers or floats, by extending the list.

2.4.4 Example 2: Parameterizing a String Length Checker

Consider a function that checks if a string exceeds a length limit:

```
def is_string_too_long(text: str, max_length: int) -> bool:
    """Return True if the string exceeds max_length."""
    return len(text) > max_length
```

Unit test:

```
import unittest
from parameterized import parameterized

class TestStringChecker(unittest.TestCase):
    @parameterized.expand([
```

```
        ("short_string", "hello", 10, False),
        ("long_string", "hello world", 5, True),
        ("equal_length", "python", 6, False),
        ("empty_string", "", 0, False),
    ])
    def test_is_string_too_long(self, name, text, max_length,
       expected):
        """Test is_string_too_long with parameterized inputs."""
        result = is_string_too_long(text, max_length)
        self.assertEqual(result, expected, f"String '{text}' with
           max_length {max_length} should return {expected}")
```

Explanation: This test defines four test cases for `is_string_too_long`, covering a short string, a long string, a string equal to the maximum length, and an empty string. The `@parameterized.expand` decorator generates a test for each case, passing the name, text, `max_length`, and expected values to the `test_is_string_too_` method. The method calls the function and verifies the result matches the expected boolean. This approach ensures comprehensive coverage of the function's behavior with minimal code, and the test names make failures easy to diagnose.

2.4.5 Example 3: Parameterizing an Exception Test

Consider a function that divides two numbers:

```
def divide(a: float, b: float) -> float:
    """Divide a by b."""
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b
```

Unit test:

```
import unittest
from parameterized import parameterized

class TestDivide(unittest.TestCase):
    @parameterized.expand([
        ("zero_denominator", 10, 0),
        ("negative_zero_denominator", 5, -0),
    ])
    def test_divide_zero_denominator(self, name, a, b):
        """Test divide raises ZeroDivisionError for zero denominator
            ."""
        with self.assertRaises(ZeroDivisionError, msg=f"Dividing {a}
            by {b} should raise ZeroDivisionError"):
            divide(a, b)
```

Explanation: This test uses parameterization to check that `divide` raises a `ZeroDivisionEr` for zero denominators. Two test cases are defined, one for a positive zero and one for a negative zero, ensuring the function handles both correctly. The `test_divide_zero`

method uses `self.assertRaises` to verify the exception is raised, with a custom message. Parameterization allows testing multiple error cases in one method, improving efficiency and readability.

2.4.6 References

- Parameterized Library Documentation: Official guide to using `parameterized.expand` https://parameterized.readthedocs.io/
- Real Python: Parameterized Testing with Pytest: Explores parameterization with pytest, relevant for comparison. https://realpython.com/pytest-python-testing/
- Python Parameterized Testing: Tutorial on parameterized testing in Python. https://python-parameterized-testing.com/
- Stack Overflow: Parameterized Testing: Community discussions on parameterization. https://stackoverflow.com/questions/tagged/parameterized-te
- TestDriven.io: Parameterized Tests in Python: Practical guide to parameterized testing. https://testdriven.io/blog/parameterized-testing-python/

2.5 Fixtures

2.5.1 Concept Overview

Fixtures provide setup and teardown code for tests, ensuring a consistent test environment. In `unittest`, fixtures are implemented using methods like `setUp`, `tearDown`, `setUpClass`, and `tearDownClass`, which prepare resources (e.g., databases, files) before tests and clean them up afterward.

2.5.2 Detailed Explanation

Fixtures are critical for tests that require resources, such as a database connection or a temporary file. The `setUp` method runs before each test, creating a fresh environment, while `tearDown` runs after each test, cleaning up to prevent interference between tests. The `setUpClass` and `tearDownClass` methods run once per test class, useful for expensive setups like initializing a server. Fixtures ensure tests are isolated and repeatable, which is especially important for integration tests involving external systems.

For beginners, fixtures simplify test writing by handling repetitive setup tasks, but they require careful management to avoid resource leaks or test dependencies. Proper fixture design ensures tests are robust and maintainable.

2.5.3 Example 1: Database Fixture

Consider testing a database operation:

```
class TestDatabase(unittest.TestCase):
    def setUp(self):
        """Set up a temporary database."""
        self.db = Database()
```

```

    def tearDown(self):
        """Clean up the database."""
        self.db.close()

    def test_save_and_retrieve(self):
        """Test saving and retrieving data."""
        data = {'id': 1, 'name': 'Alice'}
        self.db.save(data)
        result = self.db.get(1)
        self.assertEqual(result, data, "Retrieved data should match
           saved data")
```

Explanation: This test uses fixtures to manage a database connection. The `setUp` method creates a new `Database` instance before each test, ensuring a clean state. The `tearDown` method closes the database, preventing resource leaks. The `test_save_and_retrieve` method saves a dictionary and retrieves it, verifying they match. The fixtures ensure the test is isolated, as each test runs with a fresh database, and cleanup prevents interference.

2.5.4 Example 2: File Fixture

Consider testing a file-writing function:

```
def write_to_file(file_path: str, content: str) -> None:
    """Write content to a file."""
    with open(file_path, 'w') as f:
        f.write(content)
```

Unit test:

```
import unittest
import tempfile
import os

class TestFileOperations(unittest.TestCase):
    def setUp(self):
        """Set up a temporary file."""
        self.temp_file = tempfile.NamedTemporaryFile(delete=False)
        self.file_path = self.temp_file.name
        self.temp_file.close()

    def tearDown(self):
        """Remove the temporary file."""
        if os.path.exists(self.file_path):
            os.unlink(self.file_path)

    def test_write_to_file(self):
        """Test writing to a file."""
        content = "Hello, World!"
        write_to_file(self.file_path, content)
        with open(self.file_path, 'r') as f:
```

```
            result = f.read()
        self.assertEqual(result, content, "File should contain
            written content")
```

Explanation: This test uses fixtures to manage a temporary file. The `setUp` method creates a temporary file using `tempfile.NamedTemporaryFile`, storing its path. The `tearDown` method deletes the file, ensuring cleanup. The `test_write_to_fi` method writes content to the file and reads it back, verifying the content matches. The fixtures ensure each test uses a fresh file, preventing conflicts and ensuring repeatability.

2.5.5 Example 3: Class-Level Fixture for Server

Consider testing a server-based function:

```
class TestServerOperations(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        """Set up a mock server."""
        cls.server = MockServer()
        cls.server.start()

    @classmethod
    def tearDownClass(cls):
        """Shut down the mock server."""
        cls.server.stop()

    def test_server_query(self):
        """Test querying the server."""
        result = self.server.query('SELECT * FROM data')
        self.assertEqual(result, ['row1', 'row2'], "Server should
            return expected data")
```

Explanation: This test uses class-level fixtures to manage a mock server, which is expensive to start and stop. The `setUpClass` method initializes and starts the server once for all tests in the class, while `tearDownClass` stops it. The `test_server_query` method tests a query, verifying the server returns expected data. Class-level fixtures are efficient for resources shared across tests, ensuring setup and cleanup are performed only once.

2.5.6 References

- Python `unittest` Fixtures Documentation: Official guide to fixtures in `unittest`. https://docs.python.org/3/library/unittest.html#class-and-module-fixt
- Real Python: Python Testing with `unittest`: Tutorial covering fixtures. https://realpython.com/python-testing/
- Python Testing: Fixtures: Guide to using fixtures in Python. https://python-testing.readthedocs.io/en/latest/fixtures.html

- Stack Overflow: `unittest` Fixtures: Community discussions on fixtures. https://stackoverflow.com/questions/tagged/python-unittest
- Pytest Fixtures: Alternative fixture implementation for comparison. https://docs.pytest.org/en/stable/fixture.html

2.6 Memoization

2.6.1 Concept Overview

Memoization is a caching technique that stores function results to avoid redundant computations, often implemented as a decorator. In this project, the `utils.memoize` decorator caches method results based on input arguments.

2.6.2 Detailed Explanation

Memoization is used to optimize functions that are computationally expensive or called repeatedly with the same inputs. By storing results in a cache, subsequent calls with the same arguments return the cached result, improving performance. In Python, a memoization decorator wraps the function, maintaining a dictionary of inputs to outputs. The `utils.memoize` decorator in this project applies this to instance methods, caching results per object instance.

For instance, a method that computes a Fibonacci number can be memoized to avoid recalculating values for the same inputs. For beginners, memoization is a way to make code faster by ”remembering” previous results, but it requires understanding trade-offs, like increased memory usage.

2.6.3 Example 1: Memoized Property

```
def memoize(func):
    """Cache function results."""
    cache = {}
    def wrapper(*args):
        """Wrapper function"""
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

class DataProcessor:
    def compute_expensive(self) -> int:
        """Perform an expensive computation."""
        return expensive_computation()

    @memoize
    def cached_result(self) -> int:
        """Return cached result of computation."""
        return self.compute_expensive()
```

Unit test:

```python
import unittest
from unittest.mock import patch

class TestDataProcessor(unittest.TestCase):
    @patch.object(DataProcessor, 'compute_expensive')
    def test_memoize(self, mock_compute):
        """Test memoization of cached_result."""
        mock_compute.return_value = 100
        processor = DataProcessor()
        result1 = processor.cached_result()
        result2 = processor.cached_result()
        self.assertEqual(result1, 100, "First call should return computed value")
        self.assertEqual(result2, 100, "Second call should return cached value")
        self.assertEqual(mock_compute.call_count, 1, "Compute_expensive should be called once")
```

Explanation: This test verifies that the `cached_result` method is memoized correctly. The `patch` decorator mocks `compute_expensive`, setting it to return 100. The test calls `cached_result` twice, checking that both return 100 and that `compute_expensive` is called only once, confirming the result is cached. The `assertEqual` methods ensure the output is correct, and `assertEqual` on `call_count` verifies memoization, demonstrating the decorator's effectiveness in avoiding redundant computations.

2.6.4 Example 2: Memoized Recursive Function

Consider a recursive Fibonacci function:

```python
def memoize(func):
    """Cache function results."""
    cache = {}
    def wrapper(*args):
        """Wrapper function"""
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fibonacci(n: int) -> int:
    """Calculate the nth Fibonacci number."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Unit test:

```python
import unittest

```

```
class TestFibonacci(unittest.TestCase):
    def test_fibonacci(self):
        """Test memoized Fibonacci calculation."""
        result = fibonacci(10)
        self.assertEqual(result, 55, "Fibonacci of 10 should be 55")
        result_again = fibonacci(10)
        self.assertEqual(result_again, 55, "Second call should
           return cached result")
```

Explanation: This test checks that the memoized `fibonacci` function computes the 10th Fibonacci number (55) correctly. The first call to `fibonacci(10)` computes the result, which is cached by the `memoize` decorator. The second call retrieves the cached result, ensuring no recomputation. The `assertEqual` checks confirm the output is 55 for both calls, verifying correctness. The test demonstrates memoization, improving performance for recursive functions by storing intermediate results.

2.6.5 Example 3: Memoized Query

Consider a function that queries cached data:

```
def memoize(func):
    """Cache function results."""
    cache = {}
    def wrapper(*args):
        """Wrapper function"""
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

class QueryService:
    @memoize
    def query(self, key: str) -> str:
        """Query data by key."""
        return database_query(key)
```

Unit test:

```
import unittest
from unittest.mock import patch

class TestQueryService(unittest.TestCase):
    @patch('__main__.database_query')
    def test_query_memoization(self, mock_query):
        """Test memoized query method."""
        mock_query.return_value = "data"
        service = QueryService()
        result1 = service.query("key1")
        result2 = service.query("key1")
        self.assertEqual(result1, "data", "First query should return
            data")
```

```
13          self.assertEqual(result2, "data", "Second query should
               return cached data")
14          self.assertEqual(mock_query.call_count, 1, "Database query
               should be called once")
```

Explanation: This test verifies that the `query` method is memoized. The `patch` mocks `database_query`, returning "data". The test calls `query` twice with the same key, checking that both return "data" and that `database_query` is called only once. The assertions `assertEqual` confirm the output, and `assertEqual` on `call_count` verifies memoization, preventing redundant database queries. The test highlights the performance benefit of memoization for expensive operations.

2.6.6 References

- Wikipedia: Memoization: Overview of memoization concepts. https://en.wikipedia.org/wiki/Memoization
- Real Python: Python Decorators: Guide to decorators, including memoization. https://realpython.com/primer-on-python-decorators/
- Python Memoization: Tutorial on implementing memoization. https://python-memoization.com/
- Stack Overflow: Memoization in Python: Community discussions on memoization. https://stackoverflow.com/questions/tagged/python-memoization
- Python Patterns: Memoization: Design patterns for memoization. https://python-patterns.guide/python/memoization/

3 Task Implementations

Now that you've built a solid foundation with the study notes, we'll implement the project's eight tasks. Each task includes a step-by-step guide, complete code, and detailed explanations to ensure you understand the solution's design and implementation.

3.1 Task 0: Parameterize a Unit Test

Objective: Write a parameterized unit test for the `utils.access_nested_map` function to verify it returns expected values for various nested dictionary inputs and key paths.

Steps:

1. Create a Test File: Create `test_utils.py` to store unit tests for the `utils` module.
2. Import Required Modules: Import `unittest` for testing, `parameterized` for parameterization, and `utils` for the function to test.
3. Define the Test Class: Create a class `TestAccessNestedMap` inheriting from `unittest.TestCase`.

4. Define Parameterized Test Method: Use `parameterized.expand` to specify test cases with nested dictionaries, key paths, and expected outputs.
5. Write the Test Logic: Call `access_nested_map` with the provided inputs and assert the result matches the expected output using `assertEqual`.
6. Add Type Hints and Documentation: Include type annotations and docstrings for clarity and compliance with project requirements.

Code:

```
#!/usr/bin/env python3
"""Unit tests for utils.access_nested_map."""
import unittest
from unittest.mock import patch
from parameterized import parameterized
from utils import access_nested_map

class TestAccessNestedMap(unittest.TestCase):
    """Test cases for access_nested_map function."""

    @parameterized.expand([
        ({"a": "1"}, ("a",), 1),
        ({"a": {"b": "2"}}, ("a",), {"b": "2"}),
        ({"a": {"b": "2"}}, {"a", "b"}, ("a", "b"), 2}),
    ])
    def test_access_nested_map(self, nested_map: dict, path: tuple,
       expected: any) -> None:
        """Test access_nested_map returns expected values for
           various inputs."""
        result = access_nested_map(nested_map, path)
        self.assertEqual(result, expected, f"Accessing {path} in {
           nested_map} should return {expected}")
```

Explanation: This task requires testing the `access_nested_map` function, which retrieves values from a nested dictionary using a tuple of keys. The test file starts with the shebang line for portability on Ubuntu 18.04 LTS, as required. The `TestAccessNestedMap` class inherits from `unittest.TestCase`, providing access to testing methods. The `test_access_nested_map` method uses `@parameterized.expa` to define three scenarios:

- A simple key lookup (`{"a": 1}`, `("a",)`, 1), expecting the value 1.
- A nested dictionary lookup (`{"a": {"b": 2}}`, `("a",)`, `{"b": "2"}`), expecting the nested dictionary.
- A deep nested key lookup (`{"a": {"b": "2"}}`, `("a", "b")`, 2), expecting the value 2.

The test calls `access_nested_map` with `nested_map` and `path`, using `self.assertEqual` to compare the result with `expected`. The error message includes the inputs and expected output for debugging. Type hints (`dict`, `tuple`, `any`) and a docstring ensure compliance with project requirements. The test is concise yet thorough,

covering multiple cases to verify the function's ability to navigate nested structures.

3.2 Task 1: Parameterize a Unit Test (Exception Handling)

Objective: Test that `utils.access_nested_map` raises a `KeyError` for invalid key paths.

Steps:

1. Extend `TestAccessNestedMap` in `test_utils.py`.
2. Define a New Test Method: Use `@parameterized.expand` to specify test cases with invalid paths.
3. Test Exception Handling: Use `assertRaises` to check for `KeyError` and verify the exception message.
4. Add Documentation: Include docstrings and type hints.

Code:

```
    @parameterized.expand([
        ({}, ("a",), "a"),
        ({"a": "1"}, ("b",), "b"),
        ({"a": "1"}, ("a", "b"), "b"),
    ])
    def test_access_nested_map_exception(self, nested_map: dict,
       path: tuple, expected_key: str) -> None:
        """Test access_nested_map raises KeyError for invalid paths.
           """
        with self.assertRaises(KeyError) as cm:
            access_nested_map(nested_map, path)
        self.assertEqual(str(cm.exception), f"'{expected_key}'", f"
           KeyError for {path} should reference '{expected_key}'")
```

Explanation: This test method extends the `TestAccessNestedMap` class to handle error cases. The `@parameterized.expand` decorator defines three test cases:

- An empty dictionary with a path `("a",)`, expecting a `KeyError` for key `"a"`.
- A dictionary `{"a": 1}` with a path `("b",)`, expecting a `KeyError` for key `"b"`.
- A dictionary `{"a": 1}` with a path `("a", "b")`, expecting a `KeyError` for key `"b"`.

The `test_access_nested_map_exception` method uses a context manager (`with self.assertRaises(KeyError)`) to capture the `KeyError` raised by `access_nested_map`. The test verifies that the exception's message matches the expected key (e.g., `'a'`), using `self.assertEqual`. The error message in the assertion aids debugging by showing the path and expected key. Type hints and a docstring ensure clarity and compliance. This test ensures the function fails gracefully for invalid inputs, a critical aspect of robust code.

3.3 Task 2: Mock HTTP Calls

Objective: Test the `utils.get_json` function without making real HTTP requests.

Steps:

1. Add to `test_utils.py`: Create a new test class `TestGetJson`.
2. Mock `requests.get`: Use `patch` to simulate HTTP responses.
3. Parameterize Test Cases: Use `parameterized.expand` to test multiple URLs and payloads.
4. Verify Behavior: Check the function's output and mock calls.
5. Add Documentation: Include type hints and docstrings.

Code:

```
#!/usr/bin/env python3
"""Unit tests for utils.get_json."""
import unittest
from unittest.mock import patch
from parameterized import parameterized
from utils import get_json

class TestGetJson(unittest.TestCase):
    """Test cases for get_json function."""

    @parameterized.expand([
        ("example_com", "http://example.com", {"payload": True}),
        ("holberton_io", "http://holberton.io", {"payload": False}),
    ])
    @patch('requests.get')
    def test_get_json(self, name: str, test_url: str, test_payload:
       dict, mock_get) -> None:
        """Test get_json returns expected payload without real HTTP
           calls."""
        mock_get.return_value.json.return_value = test_payload
        result = get_json(test_url)
        self.assertEqual(result, test_payload, f"get_json({test_url
           }) should return {test_payload}")
        mock_get.assert_called_once_with(test_url)
```

Explanation: This test verifies that `get_json` correctly processes JSON responses without network calls. The `TestGetJson` class uses `@patch('requests.get')` to mock the `requests.get` function, ensuring no real HTTP requests are made. The `parameterized.expand` decorator defines two test cases with different URLs and payloads, identified by names for clarity. The `mock_get` object is configured to return a mock response whose `json()` method yields `test_payload`. The test calls `get_json`, checking that it returns the expected payload and that `mock_get` was called once with the correct URL. The assertions ensure the function's output is correct and the mock was used appropriately. Type hints and a docstring

provide clarity, and the shebang line ensures compatibility with Ubuntu 18.04 LTS.

3.4 Task 3: Parameterize and Patch

Objective: Test the `utils.memoize` decorator to ensure it caches method results.

Steps:

1. Add to `test_utils.py`: Create a `TestMemoize` class.
2. Define a Test Class: Create an inner class with a method and memoized property.
3. Mock the Method: Use `patch.object` to mock the underlying method.
4. Verify Caching: Check that the property returns the same result and the method is called once.
5. Add Documentation: Include docstrings and type hints.

Code:

```
#!/usr/bin/env python3
"""Unit tests for utils.memoize."""
import unittest
from unittest.mock import patch
from utils import memoize

class TestMemoize(unittest.TestCase):
    """Test cases for memoize decorator."""

    def test_memoize(self):
        """Test memoize caches method calls."""
        class TestClass:
            def a_method(self):
                """Return a value."""
                return 42

            @memoize
            def a_property(self):
                """Return cached result of a_method."""
                return self.a_method()

        with patch.object(TestClass, 'a_method') as mock_method:
            mock_method.return_value = 42
            obj = TestClass()
            result1 = obj.a_property()
            result2 = obj.a_property()
            self.assertEqual(result1, 42, "First call should return
                42")
            self.assertEqual(result2, 42, "Second call should return
                cached 42")
```

```
            mock_method.assert_called_once()
```

Explanation: This test ensures the `memoize` decorator caches results of the `a_property` method. The `TestMemoize` class defines an inner `TestClass` with `a_method` (returning 42) and `a_property`, decorated with `memoize`. The `patch.object` context manager mocks `a_method`, setting its return value to 42. The test creates a `TestClass` instance and calls `a_property` twice, verifying both return 42. The `assert_called_once` method confirms `a_method` was called only once, proving the second call used the cached result. The assertions include messages for debugging, and the docstring and type hints clarify the test's purpose. This test demonstrates the decorator's ability to optimize performance by storing results.

3.5 Task 4: Parameterize and Patch as Decorators

Objective: Test the `GithubOrgClient.org` property without HTTP calls.

Steps:

1. Create Test File: Create `test_client.py` for `GithubOrgClient` tests.
2. Import Modules: Include `unittest`, `parameterized`, `unittest`, and `client`.
3. Create Test Class: Define `TestGithubOrgClient` inheriting from `unittest.TestCase`.
4. Mock `get_json`: Use `@patch` to mock `client.get_json`.
5. Parameterize Test Cases: Use `parameterized.expand` to test multiple organizations.
6. Verify Behavior: Check the property's output and mock calls.
7. Add Documentation: Include type hints and docstrings.

Code:

```python
#!/usr/bin/env python3
"""Unit tests for client.GithubOrgClient."""
import unittest
from unittest.mock import patch
from parameterized import parameterized
from client import GithubOrgClient

class TestGithubOrgClient(unittest.TestCase):
    """Test cases for GithubOrgClient."""

    @parameterized.expand([
        ("google_org", "google", {"org": "google_data"}),
        ("abc_org", "abc", {"org": "abc_data"}),
    ])
    @patch('client.get_json')
    def test_org(self, name: str, org_name: str, expected: dict,
       mock_get_json) -> None:
        """Test GithubOrgClient.org returns correct value without
           HTTP calls."""
        mock_get_json.return_value = expected
```

```
        client = GithubOrgClient(org_name)
        result = client.org()
        self.assertEqual(result, expected, f"org for {org_name}
            should return {expected}")
        mock_get_json.assert_called_once_with(f"https://api.github.
            com/orgs/{org_name}")
```

Explanation: This test verifies the `GithubOrgClient.org` property, which fetches organization data from a URL. The `TestGithubOrgClient` class uses `@patch('client.get_` to mock the `get_json` function, preventing real HTTP requests. The `parameterized.expand` decorator defines two test cases for organizations `"google"` and `"abc"`, each with a mocked payload. The test creates a `GithubOrgClient` instance, calls `org`, and checks that it returns the expected payload. The `assert_called_once_with` ensures `get_json` was called with the correct API URL. The test is robust, covering multiple organizations, and includes type hints, a docstring, and a shebang line for compliance.

3.6 Task 5: Mocking a Property

Objective: Test the `GithubOrgClient._public_repos_url` property.

Steps:

1. Extend `TestGithubOrgClient` in `test_client.py`.
2. Mock `org` Property: Use `patch` with `PropertyMock` to simulate `org`.
3. Add Test Method: Verify the property returns the expected URL.
4. Add Documentation: Include docstrings and type hints.

Code:

```
    def test_public_repos_url(self):
        """Test GithubOrgClient._public_repos_url."""
        with patch('client.GithubOrgClient.org', new_callable=
            PropertyMock) as mock_org:
            mock_org.return_value = {"repos_url": "https://api.
                github.com/orgs/test/repos"}
            client = GithubOrgClient("test")
            result = client._public_repos_url
            self.assertEqual(result, "https://api.github.com/orgs/
                test/repos", "Should return repos URL")
```

Explanation: This test checks that the `_public_repos_url` property extracts the `repos_url` from the `org` property's data. The `patch` with `new_callable=PropertyMock` mocks the `org` property, setting it to return a dictionary with a `repos_url` key. The test creates a `GithubOrgClient` instance for `"test"` and accesses `_public_repos_url`, verifying it returns the expected URL. The assertion includes a message for clarity. The test is focused, ensuring the property correctly processes the mocked data, and includes a docstring for documentation.

3.7 Task 6: More Patching

Objective: Test the `GithubOrgClient.public_repos` method.

Steps:

1. Extend `TestGithubOrgClient`: Add to `test_client.py`.
2. Mock Dependencies: Use `@patch` for `get_json` and `patch` for `_public_repos_url`.
3. Add Test Method: Verify the method returns a list of repo names.
4. Add Documentation: Include docstrings and type hints.

Code:

```
    @patch('client.get_json')
    def test_public_repos(self, mock_get_json):
        """Test GithubOrgClient.public_repos."""
        test_payload = [{"name": "repo1"}, {"name": "repo2"}]
        mock_get_json.return_value = test_payload
        with patch('client.GithubOrgClient._public_repos_url',
           new_callable=PropertyMock) as mock_url:
            mock_url.return_value = "https://api.github.com/orgs/
               test/repos"
            client = GithubOrgClient("test")
            result = client.public_repos()
            self.assertEqual(result, ["repo1", "repo2"], "Should
               return list of repo names")
            mock_url.assert_called_once()
            mock_get_json.assert_called_once_with("https://api.
               github.com/orgs/test/repos")
```

Explanation: This test verifies that `public_repos` returns a list of repository names. The `@patch('client.get_json')` mocks `get_json`, returning a list of repository dictionaries. A nested `patch` with `PropertyMock` mocks `_public_repos_url`, setting it to a test URL. The test creates a client, calls `public_repos`, and checks that it returns `["repo1", "repo2"]`. Assertions `assert_called_once` and `assert_called` ensure the mocks were called correctly. The test covers the method's core functionality, with a docstring and type hints for clarity.

3.8 Task 7: Parameterize

Objective: Test the `GithubOrgClient.has_license` method.

Steps:

1. Extend `TestGithubOrgClient`: Add to `test_client.py`.
2. Define Parameterized Test: Use `@parameterized.expand` for license checks.
3. Verify Behavior: Check the method's boolean output.
4. Add Documentation: Include docstrings and type hints.

Code:

```
    @parameterized.expand([
        ("valid_license", {"license": {"key": "my_license"}}, "
           my_license", True),
        ("invalid_license", {"license": {"key": "other_license"}}, "
           my_license", False),
    ])
    def test_has_license(self, name, repo: dict, license_key: str,
       expected: bool) -> None:
        """Test GithubOrgClient.has_license."""
        client = GithubOrgClient("test")
        result = client.has_license(repo, license_key)
        self.assertEqual(result, expected, f"has_license({repo}, {
           license_key}) should return {expected}")
```

Explanation: This test checks that `has_license` correctly identifies whether a repository has a specific license. The `@parameterized.expand` decorator defines two cases: one where the repository's license matches the key (`True`), and one where it doesn't (`False`). The test creates a client, calls `has_license`, and verifies the result matches the expected boolean. The assertion includes a message for debugging, and the docstring and type hints ensure clarity. The test is efficient, using parameterization to cover multiple scenarios.

3.9 Task 8: Integration Test: Fixtures

Objective: Test the integration of `GithubOrgClient.public_repos` using fixtures.

Steps:

1. Extend `test_client.py`: Create `TestIntegrationGithubOrgClient`.
2. Use Parameterized Fixtures: Apply `@parameterized_class` with fixture data.
3. Mock HTTP Requests: Use `patch` with `side_effect` for API responses.
4. Add Test Method: Verify the repository list.
5. Add Fixtures: Use `setUpClass` and `tearDownClass` for setup.
6. Add Documentation: Include docstrings and type hints.

Code:

```
from parameterized import parameterized_class
from unittest.mock import patch, Mock

@parameterized_class([
    {
        "org_payload": {"repos_url": "https://api.github.com/orgs/
           test/repos"},
        "repos_payload": [{"name": "repo1"}, {"name": "repo2"}],
        "expected_repos": ["repo1", "repo2"],
        "apache2_repos": ["repo1"]
```

```
    }
])
class TestIntegrationGithubOrgClient(unittest.TestCase):
    """Integration tests for GithubOrgClient.public_repos."""

    @classmethod
    def setUpClass(cls):
        """Set up mocks for HTTP requests."""
        cls.get_patcher = patch('requests.get')
        cls.mock_get = cls.get_patcher.start()
        cls.mock_get.side_effect = [
            Mock(json=Mock(return_value=cls.org_payload)),
            Mock(json=Mock(return_value=cls.repos_payload))
        ]

    @classmethod
    def tearDownClass(cls):
        """Stop the patcher."""
        cls.get_patcher.stop()

    def test_public_repos(self):
        """Test public_repos integration."""
        client = GithubOrgClient("test")
        result = client.public_repos()
        self.assertEqual(result, self.expected_repos, "public_repos
           should return expected repos")
```

Explanation: This integration test verifies that `public_repos` fetches and processes repository data correctly. The `@parameterized_class` decorator uses fixtures to define test data, including `org_payload` and `repos_payload`. The `setUpClass` method patches `requests.get`, using `side_effect` to return mock responses for the organization and repository APIs. The `tearDownClass` method stops the patcher. The `test_public_repos` method creates a client, calls `public_repos`, and checks that it returns the expected list. The assertion ensures the output matches `expected_repos`, with a message for clarity. The test exercises the full code path, with mocked HTTP calls, ensuring integration works as intended.

4 Repository Information

GitHub Repository: https://github.com/Alxai/python-testing

Directory: `0x03-unittests_and_integration_tests`

Files: `test_utils.py`, `test_client.py`

5 Copyright