1 Task Implementations

1.1 Task 0: Parameterize a Unit Test

1.1.1 Concept: Parameterized Testing

Parameterized testing enables a single test method to run with multiple input-output pairs, ensuring a functions behavior is validated across diverse scenarios without duplicating code. This is critical for functions handling varied inputs, such as different data structures or edge cases. The `parameterized` librarys `parameterized.expand` decorator takes a list of tuples, each containing test arguments and expected results. This approach minimizes boilerplate, enhances maintainability, and isolates failures to specific input sets, simplifying debugging. Parameterized testing is ideal for testing functions with multiple use cases, as it allows adding new cases by extending the parameter list, ensuring comprehensive coverage.

For example, a function that computes the square of a number should be tested with positive, negative, and zero inputs. Parameterized testing consolidates these cases into one method, making the test suite efficient and scalable.

Example: Testing a square function:

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

class TestSquare(unittest.TestCase):
    @parameterized.expand([
        (2, 4),     # Positive number
        (-3, 9),    # Negative number
        (0, 0),     # Edge case: zero
    ])
    def test_square(self, input_num, expected):
        # Verify square returns expected result
        self.assertEqual(square(input_num), expected)
```

This runs three test cases in one method, covering key scenarios.

1.1.2 Steps to Implement

1. **Understand Function**: $\texttt{access}_{n}ested_{m}apretrievesavaluefromanesteddictionaryusingatupleofk$
$*ChooseTestCases** : Selectinputsfor : -Single-leveldictionary(\texttt{\{"a": 1\}}, path(\texttt{"a",})).-$
$Nesteddictionary, firstlevel(\texttt{\{"a": \{"b": 2\}\}}, path(\texttt{"a",})).-Nesteddictionary, deepervalue(\texttt{\{"a'}$
$*SetUpTestFile** : Create\texttt{test}_{u}tils.pywithimportsfor\texttt{unittest}, \texttt{parameterized}, and\texttt{access}_{n}ested_{m}$

```
#!/usr/bin/env python3
"""Unit tests for utils.access_nested_map."""

from parameterized import parameterized  # Parameterize test
    cases
from utils import access_nested_map  # Function to test:
    navigates nested dict
import unittest  # Unit testing framework

```

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

    @parameterized.expand([
        # Test case 1: Single key-value pair
        ({"a": 1}, ("a",), 1),
        # Test case 2: Nested dict, access first level
        ({"a": {"b": 2}}, ("a",), {"b": 2}),
        # Test case 3: Nested dict, access deeper value
        ({"a": {"b": 2}}, ("a", "b"), 2),
    ])
    def test_access_nested_map(self, nested_map, path, expected):
        """Test access_nested_map retrieves correct value.

        Args:
            nested_map (dict): Input dictionary.
            path (tuple): Key path to navigate.
            expected: Expected value at path.
        """
        # Call function with test inputs
        result = access_nested_map(nested_map, path)
        # Verify result matches expected output
        self.assertEqual(result, expected)
```

Listing 1: $\text{test}_{u}tils.py : TestAccessNestedMapforTask0$

Explanation The $\text{test}_{a}ccess_{n}ested_{m}apmethodtests\text{access}_{n}ested_{m}ap, whichnavigatesanesteddiction$

1.2.1 Concept: Exception Testing

Exception testing verifies that a function raises appropriate exceptions for invalid inputs, ensuring robust error handling. This is vital for functions processing dynamic data, where errors like missing keys or invalid types are common. In unittest, the assertRaises context manager captures an expected exception, allowing the test to inspect its type and message. This ensures the function fails gracefully and provides meaningful error feedback, which is essential for debugging and user experience. Exception testing targets edge cases and error conditions, validating that the functions error handling is correct and consistent.

For example, a function that accesses a dictionary key should raise a KeyError for missing keys. The assertRaises context manager wraps the function call, catching the exception and enabling verification of its message, ensuring precise error reporting.

Example: Testing a function for missing dictionary keys:

```python
def get_value(d, key):
    if key not in d:
        raise KeyError(key)
    return d[key]

class TestGetValue(unittest.TestCase):
```

```
    def test_missing_key(self):
        # Expect KeyError for missing key
        with self.assertRaises(KeyError) as cm:
            get_value({}, "x")  # Call with missing key
        # Verify error message
        self.assertEqual(str(cm.exception), "'x'")
```

This tests that get$_{v}alueraisesa$KeyError$withthecorrectmessage.$

Steps to Implement 1. **Analyze Error Behavior**: access$_{n}ested_{m}apraises$KeyError$fornot$
$existentkeysinthepath.2.**SelectTestCases**:Choose:-Emptydictionary,path$("a",).–

$Dictionary${"a": 1}$,path$("a", "b").3.$**ExtendTestClass**:Addto$TestAccessNestedMap$intest$

1.3.1 Concept: Mocking HTTP Calls

Mocking HTTP calls isolates a functions logic from external network dependencies, which are slow, unreliable, and may incur costs. The unittest.mock.patch decorator replaces functions like requests.get with a mock object, allowing tests to control the response. This ensures tests are fast, deterministic, and independent of network conditions. Mocking is essential for API-dependent functions, as it simulates real-world responses without hitting the actual API.

To mock an HTTP call, the mocks return$_{v}alueisconfiguredtomimictheresponseobjectsstructure,$

1.4.1 Concept: Memoization

Memoization caches a functions result to avoid redundant computations, optimizing performance for expensive operations. The memoize decorator in utils.py turns a method into a property, storing its result in an instance attribute after the first call. Subsequent accesses retrieve the cached value, bypassing the method. This is useful for methods involving heavy computations or external calls, ensuring efficiency in object-oriented code.

Testing memoization involves mocking the underlying method to track call counts and verify its called only once. The unittest.mock.patch function enables this, and assert$_{c}alled_{o}nceconfirmsthecachingbehavior.Thisensuresthedecoratorworksasintended$

Example: Testing a memoized calculation:

```
from functools import wraps
def memoize(fn):
    attr_name = f"_{fn.__name__}"
    @wraps(fn)
    def memoized(self):
```

```
        if not hasattr(self, attr_name):
            setattr(self, attr_name, fn(self))
        return getattr(self, attr_name)
    return property(memoized)

class Math:
    @memoize
    def calculate(self):
        return 50  # Simulate heavy computation

class TestMath(unittest.TestCase):
    def test_memoize(self):
        # Mock calculate method
        with patch.object(Math, 'calculate', return_value=50) as
           mock_calc:
            math = Math()
            # Access property twice
            result1 = math.calculate
            result2 = math.calculate
            # Verify results
            self.assertEqual(result1, 50)
            self.assertEqual(result2, 50)
            # Verify single call
            mock_calc.assert_called_once()
```

This confirms calculate is called once and cached.

Steps to Implement 1. **Understand Decorator**: memoize caches method results as a property. 2. **Design Test**: Create a class with a mockable method and memoized property. 3. **Set Up Test Class**: Add TestMemoize to test$_{u}$*tils.py*.4.*

$*DefineNestedClass** : CreateTestClasswitha_methodanda_property.5. **MockMethod** : Usepat$

1.5.1 Concept: Mocking with Decorators

The @patch decorator mocks module-level functions, injecting a mock object into the test method, simplifying setup compared to context managers. Combined with parameterized.expand, it enables testing multiple scenarios without external dependencies, ideal for API-dependent code. The decorator approach reduces code nesting, enhancing readability, while parameterization ensures comprehensive input coverage. The mocks $assert_called_once_with verifies correct arguments, ensuring proper depe$

This technique is powerful for methods calling external functions, as it isolates logic while testing various inputs. For example, a method fetching user data can be tested with different usernames, mocking the API call to simulate responses.

Example: Testing a user data fetcher:

```
def fetch_user(user_id):
    return get_json(f"https://api.example.com/users/{user_id}")

```

```python
class TestFetchUser(unittest.TestCase):
    @patch('__main__.get_json')  # Mock get_json
    @parameterized.expand(['user1', 'user2'])  # Multiple users
    def test_fetch_user(self, user_id, mock_get_json):
        # Set mock response
        mock_get_json.return_value = {"id": user_id}
        # Call function
        result = fetch_user(user_id)
        # Verify result
        self.assertEqual(result, {"id": user_id})
        # Verify URL
        mock_get_json.assert_called_once_with(f"https://api.example.com/user
```

Steps to Implement 1. **Analyze Property**: GithubOrgClient.org calls get$_{j}sonfororg$
$*ChooseTestCases**:Testgoogleandabcorganizations.3.**SetUpTestFile**:$

$Create$`test`$_{c}lient.pywithimports.4. * *DefineTestClass * * : Add$`TestGithubOrgClient`$.5. * *Mockg$

1.6.1 Concept: Mocking Properties

Mocking a property replaces its getter with a mock object, controlling its output without running the real implementation. Using unittest.mock.PropertyMock with patch, tests can simulate properties that depend on external data or complex logic. This isolates the propertys behavior, focusing on how its used by other methods. The mocks return$_{v}alueissettothedesiredoutput, andthetestverifiescorrectaccess, mainta$

This is crucial for properties like those fetching API data, ensuring tests dont rely on external systems. For example, mocking a property that retrieves a URL allows testing dependent methods without network calls.

Example: Testing a URL property:

```python
class ApiClient:
    @property
    def endpoint(self):
        return get_json("https://api.example.com/config")["url"]

class TestApiClient(unittest.TestCase):
    def test_endpoint(self):
        # Mock endpoint property
        with patch('__main__.ApiClient.endpoint',
           new_callable=PropertyMock) as mock_endpoint:
            # Set mock output
            mock_endpoint.return_value = "https://test.com"
            client = ApiClient()
            # Verify result
            self.assertEqual(client.endpoint, "https://test.com")
```

Steps to Implement 1. **Analyze Property**: GithubOrgClient.$_{p}ublic_{r}epos_{u}rlgetsrepos_{u}rlj$

1.7.1 Concept: Multiple Mocks

Multiple mocks test methods with several dependencies, such as properties and external functions. The @patch decorator mocks module-level functions, while patch context managers handle properties. This isolates the methods logic, ensuring tests focus on its behavior. The assert$_{c}alled_{o}ncemethodverifieseachmocksintera$

This is ideal for methods chaining dependencies, like fetching data using a property-defined URL. Mocking both ensures the test simulates the workflow without real calls.

Example: Testing a data fetcher:

```
class Fetcher:
    @property
    def url(self):
        return "https://api.example.com/data"

    def fetch(self):
        return [item["name"] for item in get_json(self.url)]

class TestFetcher(unittest.TestCase):
    @patch('__main__.get_json')  # Mock get_json
    def test_fetch(self, mock_get_json):
        # Mock url property
        with patch('__main__.Fetcher.url',
           new_callable=PropertyMock) as mock_url:
            # Configure mocks
            mock_url.return_value = "https://test.com"
            mock_get_json.return_value = [{"name": "a"},
               {"name": "b"}]
            fetcher = Fetcher()
            # Verify result
            self.assertEqual(fetcher.fetch(), ["a", "b"])
            # Verify calls
            mock_get_json.assert_called_once()
            mock_url.assert_called_once()
```

Steps to Implement 1. **Analyze Method**: GithubOrgClient.public$_r$epos$_u$ses$_p$ublic$_r$epos$_u$r

1.8.1 Concept: Static Method Testing

Static methods operate at the class level, independent of instance state, and are tested like regular functions. Parameterization with parameterized.expand covers multiple input scenarios, ensuring robustness. This is efficient for utility methods, as it consolidates test cases into one method, making the test suite scalable.

For example, a static method checking user permissions can be tested with different user roles, ensuring all cases are handled correctly.

Example: Testing a permission checker:

```
class Auth:
    @staticmethod
    def has_permission(user, perm):
        return user.get("permission") == perm

class TestAuth(unittest.TestCase):
    @parameterized.expand([
        ({"permission": "admin"}, "admin", True),  # Has
           permission
```

```
        ({"permission": "user"}, "admin", False),  # No
            permission
    ])
    def test_has_permission(self, user, perm, expected):
        # Verify result
        self.assertEqual(Auth.has_permission(user, perm),
            expected)
```

Steps to Implement 1. **Analyze Method**: GithubOrgClient.has$_{l}icensechecks licensekey.2.
$*ChooseTestCases**: Matchingandnon-matchinglicenses.3.**ExtendTestClass**:$
$Addto$TestGithubOrgClient.4. $**Parameterize**: Use$@parameterized.expand.5. $*

$*WriteTest**: Verify$has$_{l}icenseoutput.6. **CommentCode**: Explaintestcasesandassertions.$ C

1.9.1 Concept: Integration Testing with Fixtures

Integration testing verifies component interactions, mocking only external calls to exercise the full code path. Fixtures provide predefined data to simulate API responses, ensuring reproducibility. The parameterized$_{c}lassdecoratorapplie$

For example, a client fetching and processing API data uses fixtures to simulate responses, testing the entire workflow.

Example: Testing a data client:

```
class DataClient:
    def get_data(self):
        return [item["id"] for item in
            get_json("https://api.example.com/data")]

TEST_FIXTURES = [([{"id": 1}, {"id": 2}], [1, 2])]
```

```

@parameterized_class([{"api_data": TEST_FIXTURES[0][0],
   "expected": TEST_FIXTURES[0][1]}])
class TestDataClient(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.get_patcher = patch('__main__.get_json')
        cls.mock_get = cls.get_patcher.start()
        cls.mock_get.return_value = cls.api_data

    @classmethod
    def tearDownClass(cls):
        cls.get_patcher.stop()

    def test_get_data(self):
        client = DataClient()
        self.assertEqual(client.get_data(), self.expected)
```

Steps to Implement 1. **Analyze Method**: GithubOrgClient.public$_{r}epos fetches repositori$

$*UseFixtures** : Load\texttt{TEST}_PAY\ LOAD\ for\ organdrepodata.3.**CreateTestClass** : Add\texttt{TestInte}$