ПРАКТИЧНА РОБОТА 09

Основи модульного тестування коду мовою Python

Система оцінювання

№	Тема	К-ть балів
1.	Туторіал 1	1,5
2.	Туторіал 2	1,5
3.	Туторіали з глав 1-2	2
	Всього за практичну роботу	**5**
4.	ІНДЗ (туторіали з глав 3-7)	3
	Всього	**8**

Виконайте інструкції, представлені в завданнях:

1. ***(Модульне тестування).*** Модуль **unittest** – це фреймворк для тестування, створення якого надихалось JUnit. Нині він підтримує автоматизацію тестування, шеринг установчого та закриваючого коду (setup and shutdown code), агрегацію тестів та незалежність тестів від фреймворку звітування (reporting framework).
   Фреймворки unittest підтримують наступні поняття:
   - Test Fixture – це код, який використовується для налаштування установок тесту, щоб він міг запуститись, а також установок закінчення, коли тест завершено. Наприклад, може знадобитись створення тимчасової бази даних до запуску тесту та її знищення після спрацювання тесту.
   - Test Case – це тестовий випадок, тобто сам тест. Зазвичай перевіряється (assert), що конкретна відповідь отримується після роботи з конкретним набором вхідних даних. Фреймворк unittest постачає базовий клас TestCase, який можна використовувати для створення нових тест-кейсів.
   - Test Suite – тестовий набір, колекція тестових випадків, інших тестових наборів чи їх обох.
   - Test Runner – запускалка, яка контролює чи координує виконання тестів або тестових наборів. Також вона надає вивід користувачу (пройдено тест чи провалено). Запускалка (runner) може використовувати графічний інтерфейс користувача чи бути простим текстовим інтерфейсом.

   Створіть невеликий модуль з назвою mymath.py та додайте в нього наступний код:

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


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


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


def divide(numerator, denominator):
    return float(numerator) / denominator
```

Представлені функції не виконують перевірку помилок і взагалі багато чого, що від них очікується. Наприклад, виклик функції add для двох рядків має повернути конкатенований рядок. Проте дане спрощення здійснено в ілюстративних цілях.
Створіть модуль test_mymath.py, який буде містити код, представлений нижче, та буде знаходитись у тому ж місці, що й mymath.py.

```
import mymath
import unittest

class TestAdd(unittest.TestCase):
    """
    Test the add function from the mymath library
    """

    def test_add_integers(self):
        """
        Test that the addition of two integers returns the correct total
        """
        result = mymath.add(1, 2)
        self.assertEqual(result, 3)

    def test_add_floats(self):
        """
        Test that the addition of two floats returns the correct result
        """
        result = mymath.add(10.5, 2)
        self.assertEqual(result, 12.5)

    def test_add_strings(self):
        """
        Test the addition of two strings returns the two string as one
        concatenated string
        """
        result = mymath.add('abc', 'def')
        self.assertEqual(result, 'abcdef')

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

Спочатку імпортуємо модулі mymath (код, який тестується) та unittest (бібліотека для тестування). Далі субкласуємо TestCase та додаємо три тести, зібрані у відповідні методи. Перша функція тестує додавання двох цілих чисел, друга – додавання двох чисел з плаваючою крапкою, остання – конкатенацію двох рядків.

Важливо, щоб назви методів починались із test, - це говорить запускалці, які методи є тестами. Кожен тестовий метод має принаймні один оператор assert, який перевірятиме отриманий результат з очікуваним. Модуль unittest підтримує багато різних типів тверджень (assert). Можна тестувати на виключення, булеві умови та багато інших видів умов.

Спробуйте запустити тест. Відкрийте термінал та перейдіть у папку, що містить модуль mymath та тестовий модуль:

```
python test_mymath.py
```

Виведе три крапки, кожна з яких говорить, що тест було пройдено.

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Можна зробити вивід більш детальним, застосовуючи прапорець -v:

```
python test_mymath.py -v
```

Тоді вивід буде на зразок такого:

```
test_add_floats (__main__.TestAdd) ... ok
test_add_integers (__main__.TestAdd) ... ok
test_add_strings (__main__.TestAdd) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Інтерфейс командного рядка. Модуль unittest постачається з кількома командами, які можуть бути корисними. Щоб дізнатись про них, можна напряму використати прапорець -h:

```
python -m unittest -h
```

Розгляньте та додайте у звіт скриншот виконання даної команди.

Збережіть версію файлу test_mymath2.py, в якій буде видалено два останніх рядка. Запустіть її за допомогою наступної команди:

```
python -m unittest test_mymath2.py
```

Результат роботи має не відрізнятись від тестування попереднього прикладу. Ми можемо використовувати модуль unittest у командному рядку, наприклад:

```
python -m unittest test_mymath2.TestAdd.test_add_integers
```

Запустіть її у себе, результат виводу має бути схожим на наступний:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Якщо є маємо кілька тестових випадків у модулі, можна викликати окремий з них, зокрема так:

```
python -m unittest test_mymath2.TestAdd
```

Більш складні тести. Напишіть частину коду, яка залежить від існування бази даних. Це буде простий скрипт, який зможе створити (якщо до цього її не було) базу з деякими початковими даними за допомогою кількох функцій, які дозволятимуть здійснювати запит, видаляти та оновлювати рядки. Назвіть скрипт simple_db.py. Необхідний код представлено нижче:

```
import sqlite3

def create_database():
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()
    # створюємо таблицю
    cursor.execute("""CREATE TABLE albums
                          (title text, artist text, release_date text,
                           publisher text, media_type text)
                       """)
    # вставляємо деякі дані
    cursor.execute("INSERT INTO albums VALUES "
                   "('Glow', 'Andy Hunter', '7/24/2012',"
                   "'Xplore Records', 'MP3')")
     # зберігаємо дані в базі
    conn.commit()

    # вставляємо кілька записів за допомогою більш безпечного методу "?"
    albums = [('Exodus', 'Andy Hunter', '7/9/2002',
               'Sparrow Records', 'CD'),
              ('Until We Have Faces', 'Red', '2/1/2011',
```

```
                'Essential Records', 'CD'),
               ('The End is Where We Begin', 'Thousand Foot Krutch',
                '4/17/2012', 'TFKmusic', 'CD'),
               ('The Good Life', 'Trip Lee', '4/10/2012',
                'Reach Records', 'CD')]
    cursor.executemany("INSERT INTO albums VALUES (?,?,?,?,?)",
                       albums)
    conn.commit()

def delete_artist(artist):
    """
    Delete an artist from the database
    """
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()
    sql = """
    DELETE FROM albums
    WHERE artist = ?
    """
    cursor.execute(sql, [(artist)])
    conn.commit()
    cursor.close()
    conn.close()
def update_artist(artist, new_name):
    """
    Update the artist name
    """
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    sql = """
    UPDATE albums
    SET artist = ?
    WHERE artist = ?
    """
    cursor.execute(sql, (new_name, artist))
    conn.commit()
    cursor.close()
    conn.close()
```

```
 def select_all_albums(artist):
    """
    Query the database for all the albums by a particular artist
    """
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    sql = "SELECT * FROM albums WHERE artist=?"
    cursor.execute(sql, [(artist)])
    result = cursor.fetchall()
    cursor.close()
    conn.close()
    return result

if __name__ == '__main__':
    import os
    if not os.path.exists("mydatabase.db"):
        create_database()

    delete_artist('Andy Hunter')
    update_artist('Red', 'Redder')
    print(select_all_albums('Thousand Foot Krutch'))
```

Очевидно, що створювати та знищувати базу даних для кожного тесту досить затратно. Проте для цілей тестування іноді потрібно робити саме так. Крім того, зазвичай не потрібно створювати повністю наповнену базу даних просто для перевірки роботи.

Модуль unittest дозволяє переозначити методи setUp() і tearDown(). Створіть метод setUp(), який створить БД та метод tearDown(), який видалятиме її після завершення тесту. Зауважте, що налаштування та видалення будуть відбуватись для кожного тесту. Це усуває можливість одному тесту так змінити базу даних, щоб якийсь із наступних тестів провалився. Перша частина класу для тестового випадку матиме вигляд:

```
import os
import simple_db
import sqlite3
import unittest

class TestMusicDatabase(unittest.TestCase):
    """
    Тестування музичної бази даних
```

```
    """

    def setUp(self):
        """
        Налаштувати тимчасову базу даних
        """
        conn = sqlite3.connect("mydatabase.db")
        cursor = conn.cursor()
        # створити таблицю
        cursor.execute("""CREATE TABLE albums
                          (title text, artist text, release_date text,
                           publisher text, media_type text)
                       """)
        # вставити деякі дані
        cursor.execute("INSERT INTO albums VALUES "
                       "('Glow', 'Andy Hunter', '7/24/2012',"
                       "'Xplore Records', 'MP3')")
         # зберегти дані в БД
        conn.commit()
         # вставити кілька записів, використовуючи безпечний метод "?"
        albums = [('Exodus', 'Andy Hunter', '7/9/2002',
                   'Sparrow Records', 'CD'),
                  ('Until We Have Faces', 'Red', '2/1/2011',
                   'Essential Records', 'CD'),
                  ('The End is Where We Begin', 'Thousand Foot Krutch',
                   '4/17/2012', 'TFKmusic', 'CD'),
                  ('The Good Life', 'Trip Lee', '4/10/2012',
                   'Reach Records', 'CD')]
        cursor.executemany("INSERT INTO albums VALUES (?,?,?,?,?)",
                           albums)
        conn.commit()

    def tearDown(self):
        """
        Видалити базу даних
        """
        os.remove("mydatabase.db")
```

Метод setUp() створюватиме базу даних, а потім наповнюватиме її деякими даними. Метод tearDown() видалятиме файл з БД. Для sqlite достатньо просто видалити відповідний файл.

Додайте кілька тестів у код. Можна дописати наступні в кінець тестового класу:

```
def test_updating_artist(self):
    """
    Tests that we can successfully update an artist's name
    """
    simple_db.update_artist('Red', 'Redder')
    actual = simple_db.select_all_albums('Redder')
    expected = [('Until We Have Faces', 'Redder',
                 '2/1/2011', 'Essential Records', 'CD')]
    self.assertListEqual(expected, actual)

def test_artist_does_not_exist(self):
    """
    Test that an artist does not exist
    """
    result = simple_db.select_all_albums('Redder')
    self.assertFalse(result)
```

Перший тест оновить ім'я одного з виконавців на Redder. Далі виконується запит, щоб перевірити існування імені нового виконавця. Наступний тест перевірить, чи існує виконавець з іменем "Redder". Цього разу його не повинно бути, оскільки БД вже видалена та заново створена.

Запустіть модуль на тестування:

```
python -m unittest test_db.py
```

Повинен повернутись результат з двома успішними тестами.

Створення тестових наборів. Більшість часу, коли викликається unittest.main(), він збиратиме тестові випадки з модуля перед їх виконанням. Проте інколи тестувальник сам бажає контролювати збирання тестів. Для цього можна використовувати клас TestSuite. Наприклад, так:

```
import unittest

from test_mymath import TestAdd



def my_suite():
        suite = unittest.TestSuite()
```

```
    result = unittest.TestResult()
    suite.addTest(unittest.makeSuite(TestAdd))
    runner = unittest.TextTestRunner()
    print(runner.run(suite))

my_suite()
```

Створення власного набору – дещо заплутаний процес. Спочатку необхідно створити екземпляри TestSuite та TestResult. Клас TestResult містить тільки результати тестів. Потім викликається addTest() для набору-об'єкту. Якщо просто передати його в TestAdd, він повинен бути екземпляром TestAdd, а TestAdd необхідно також реалізувати метод runTest(). Оскільки цього не було зроблено, використовуйте функцію makeSuite() з unittest, щоб перетворити клас TestCase в набір.

Останній крок – запустити тестовий набір, для чого знадобиться запускалка. Тому створюється екземпляр TextTestRunner, який запускає набір. Результат буде приблизно таким:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
<unittest.runner.TextTestResult run=3 errors=0 failures=0>
```

У якості альтернативи можна просто викликати suite.run(result) та вивести результат. Проте отримається об'єкт TestResult, який виглядає дуже схоже на останній рядок попереднього виводу. Якщо бажаєте більш звичний вивід, використовуйте запускалку.

Пропуск тестів. Підтримується з версії Python 3.1. Існує кілька сценаріїв використання для пропуску тестів:

- Ви можете пропустити тест, якщо версія бібліотеки не підтримує те, що ви хочете протестувати.
- Тест залежить від операційної системи, в якій він працює.
- У вас є інші критерії для пропуску тесту.

Внесіть зміни в тестовий клас, щоб він мав кілька тестів, що пропускатимуться:

```
import mymath
import sys
import unittest

class TestAdd(unittest.TestCase):
    """
    Test the add function from the mymath module
    """
```

```
    def test_add_integers(self):
        """
        Test that the addition of two integers returns the correct total
        """
        result = mymath.add(1, 2)
        self.assertEqual(result, 3)

    def test_add_floats(self):
        """
        Test that the addition of two floats returns the correct result
        """
        result = mymath.add(10.5, 2)
        self.assertEqual(result, 12.5)

    @unittest.skip('Skip this test')
    def test_add_strings(self):
        """
        Test the addition of two strings returns the two string as one
        concatenated string
        """
        result = mymath.add('abc', 'def')
        self.assertEqual(result, 'abcdef')

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_adding_on_windows(self):
        result = mymath.add(1, 2)
        self.assertEqual(result, 3)
```

Далі продемонструємо два різних способи пропускати тест: skip та skipUnless. Ви помітите, що ми декоруємо функції, які потрібно пропустити. Декоратор skip може використовуватися для пропуску будь-якого тесту з будь-якої причини. Декоратор skipUnless буде пропускати тест, якщо умова не поверне True. Отже, якщо ви запустите цей тест на Mac або Linux, його буде пропущено. Існує також декоратор skipIf, який буде пропускати тест, якщо умова буде True.

Можна запустити цей скрипт з verbose-прапорцем, щоб побачити, чому тести пропускаються:

```
python -m unittest test_mymath.py -v
```

Додайте результат виводу у звіт.

Також існує декоратор expectedFailure, який можна додати до тесту, який має провалитись.

Взаємодія з doctest. Модуль unittest може використовуватись разом з модулем doctest. Якщо було створено багато модулів, які мають в собі doctest-и, зазвичай виникає потреба їх систематичного запуску. Тут на допомогу приходить unittest. Цей модуль підтримує Test Discovery, починаючи з Python 3.2. Test Discovery на базовому рівні дозволяє модулю unittest переглядати вміст папки та визначати з назви файлу, який з них може мати тести. Потім він імпортує та завантажує їх.

Створіть новий порожній каталог, а всередині нього – файл з назвою my_docs.py. Код з цього файлу має бути таким:

```
def add(a, b):
    """
    Повертає результат додавання аргументів: a + b

    >>> add(1, 2)
    3
    >>> add(-1, 10)
    9
    >>> add('a', 'b')
    'ab'
    >>> add(1, '2')
    Traceback (most recent call last):
      File "test.py", line 17, in <module>
        add(1, '2')
      File "test.py", line 14, in add
        return a + b
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
    """
    return a + b

def subtract(a, b):
    """
    Повертає результат віднімання b від a

    >>> subtract(2, 1)
    1
    >>> subtract(10, 10)
    0
    >>> subtract(7, 10)
    -3
    """
```

```
    return a - b
```

Тепер необхідно створити інший модуль у тому ж місці, який перетворить наші doctest-и в unittest-и. Назвемо цей файл test_doctests.py. Помістіть наступний код у цей файл:

```
import doctest
import my_docs
import unittest


def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_docs))
    return tests
```

Назва функції потрібна тут для роботи Test Discovery, відповідно до документації модуля doctest. Тут використовується клас DocTestSuite з модуля doctest. Можна задати для цього класу методи setup() і teardown() в якості параметрів, що можуть знадобитись тестам для роботи. Для запуску цього коду необхідно виконати наступну команду в даній папці:

```
python -m unittest discover
```

Зауважте, що коли ви використовуєте unittest для запуску doctest, кожен docstring вважається одним тестом. Якщо потрібно запустити docstring з doctest безпосередньо, то ви помітите, що doctest скаже, що існує ще кілька тестів. Крім того, він працює майже так, як очікувалося.

2. ***(Заглушки)*** Уявіть, що Ви пишете додаток з соціальною складовою та бажаєте відтестувати нову характеристику «Запостити в Facebook», проте не бажаєте в дійсності публікувати пости при кожному запуску набору тестів. Бібліотека unittest включає підпакет unittest.mock (з Python 3.3+), який пропонує потужні інструменти для мокінгу та створення заглушок у коді.
   Системні виклики є чудовими кандидатами для мокінгу, оскільки вони призводять до побічних ефектів, небажаних при тестуванні. Наприклад, при написанні скриптів для роботи з дисководом (постійні відкриття та закриття), для веб-сервера (очищатиме застарілі кешовані файли в папці /tmp) тощо. З іншого боку, продуктивні модульні тести передбачають тримання «повільного» коду за межами запусків автоматизованих тестів, зокрема для роботи з мережею або файловою системою.
   Продемонструємо це на простому прикладі видалення файлів. Напишемо функцію, яка спрощуватиме періодичне видалення файлів.

```
import os

def rm(filename):
    os.remove(filename)
```

Очевидно, що на даному етапі метод rm() не забезпечує додаткової функціональності, ніж метод `os.remove()`, проте в подальшому вона буде розширена. Запишемо традиційний тестовий випадок (без моків):

```
from mymodule import rm

import os.path
import tempfile
import unittest

class RmTestCase(unittest.TestCase):
    tmpfilepath = os.path.join(tempfile.gettempdir(), "tmp-testfile")

    def setUp(self):
        with open(self.tmpfilepath, "wb") as f:
            f.write(b"Delete me!")

    def test_rm(self):
        # remove the file
        rm(self.tmpfilepath)
```

```
        # test that it was actually removed
        self.assertFalse(os.path.isfile(self.tmpfilepath), "Failed to
remove the file.")
```

Тест досить простий, проте кожного разу тимчасовий файл створюється та видаляється. Крім цього, немає способу перевірити, чи наш метод rm() коректно передає аргумент у виклик os.remove(). Припускається, що це так, за рахунок виконання вище згаданого тесту, проте багато чого все ще треба перевірити.

Виконаємо рефакторинг (реструктуризацію) коду за допомогою mock-об’єкта:

```
from mymodule import rm

import mock
import unittest

class RmTestCase(unittest.TestCase):

    @mock.patch('mymodule.os')
    def test_rm(self, mock_os):
        rm("any path")
        # test that rm called os.remove with the right parameters
        mock_os.remove.assert_called_with("any path")
```

З даним кодом було фундаментально змінено спосіб роботи тесту. Тепер присутній інсайдер – об’єкт, який можна використати для перевірки функціональності іншого об’єкта.

Однією з перших речей, на яку слід звернути увагу, є використання декоратору методу `mock.patch()`, щоб виконати мокінг об’єкта, розташованого в модулі mymodule.os, та впровадити даний мок у тестовий метод. Зверніть увагу, що імпортування модуля створює локальну копію модуля os, тому відокремлений мокінг модуля os не спричинить впливу на модуль mymodule.

Загальна рекомендація: виконуйте мокінг об’єкта там, де цей об’єкт використовується, а не там, звідки він надходить.

Якщо потрібно виконати мокінг модуля tempfile для класу myproject.app.MyElaborateClass, ймовірно, слід застосувати мок до myproject.app.tempfile, оскільки кожен модуль має власні імпорти.

Визначений раніше метод rm дуже спрощений. Потрібно перевіряти шлях до файлу до того, як пробувати його видаляти.

```
import os
import os.path

def rm(filename):
    if os.path.isfile(filename):
        os.remove(filename)
```

Адаптуємо тестовий випадок для перевірки такої ситуації:

```
from mymodule import rm

import mock
import unittest

class RmTestCase(unittest.TestCase):

    @mock.patch('mymodule.os.path')
    @mock.patch('mymodule.os')
    def test_rm(self, mock_os, mock_path):
        # set up the mock
        mock_path.isfile.return_value = False

        rm("any path")

        # test that the remove call was NOT called.
        self.assertFalse(mock_os.remove.called, "Failed to not remove the
file if not present.")

        # make the file 'exist'
        mock_path.isfile.return_value = True

        rm("any path")

            mock_os.remove.assert_called_with("any path")
```

Парадигма тестування змінилась – тепер можна перевіряти та проводити валідацію внутрішньої функціональності методів без ***жодних*** сторонніх ефектів.
Поки що ми працювали, постачаючи мок-об'єкти для функцій, проте не для методів об'єктів чи класів, де мокінг потрібний для надсилання параметрів. Розпочнемо з рефакторингу методу rm() у службовому класі. Необхідності в інкапсуляції такої простої функції немає, проте для демонстрації концепції моків зробимо це.

```
import os
import os.path

class RemovalService(object):
    """A service for removing objects from the filesystem."""

    def rm(self, filename):
        if os.path.isfile(filename):
            os.remove(filename)
```

Зауважте, що в коді тестового випадку теж небагато що змінилось:

```
from mymodule import RemovalService
import mock
import unittest

class RemovalServiceTestCase(unittest.TestCase):
    @mock.patch('mymodule.os.path')
    @mock.patch('mymodule.os')
    def test_rm(self, mock_os, mock_path):
        # instantiate our service
        reference = RemovalService()

        # set up the mock
        mock_path.isfile.return_value = False

        reference.rm("any path")

        # test that the remove call was NOT called.
        self.assertFalse(mock_os.remove.called, "Failed to not remove the
file if not present.")
```

```
        # make the file 'exist'
        mock_path.isfile.return_value = True

        reference.rm("any path")

        mock_os.remove.assert_called_with("any path")
```

Тепер ми знаємо, що клас RemovalService працює так, як було передбачено. Створимо ще одну службу, яка оголошує його в якості залежності (dependency):

```
import os
import os.path

class RemovalService(object):
    """A service for removing objects from the filesystem."""
    def rm(self, filename):
        if os.path.isfile(filename):
            os.remove(filename)


class UploadService(object):
    def __init__(self, removal_service):
        self.removal_service = removal_service

    def upload_complete(self, filename):
        self.removal_service.rm(filename)
```

Оскільки клас RemovalService уже покритий тестами, не будемо виконувати валідацію внутрішньої функціональності методу rm() в тестах класу UploaadService. Краще відтестувати виклик методу RemovalService.rm() в класі UploadService. Для цього може бути два способи:

1) Мокувати власне метод RemovalService.rm();
2) Ввести мокований екземпляр в конструктор класу UploadService.

СПОСІБ 1. МОКУВАННЯ МЕТОДІВ ЕКЗЕМПЛЯРУ

Бібліотека mock має спеціальний декоратор методу для мокування методів об’єкту та властивостей - @mock.patch.object:

```
from mymodule import RemovalService, UploadService
```

```
import mock
import unittest

class RemovalServiceTestCase(unittest.TestCase):
    @mock.patch('mymodule.os.path')
    @mock.patch('mymodule.os')
    def test_rm(self, mock_os, mock_path):
        # instantiate our service
        reference = RemovalService()

        # set up the mock
        mock_path.isfile.return_value = False

        reference.rm("any path")

        # test that the remove call was NOT called.
        self.assertFalse(mock_os.remove.called, "Failed to not remove the
file if not present.")

        # make the file 'exist'
        mock_path.isfile.return_value = True

        reference.rm("any path")

        mock_os.remove.assert_called_with("any path")


class UploadServiceTestCase(unittest.TestCase):
    @mock.patch.object(RemovalService, 'rm')
    def test_upload_complete(self, mock_rm):
        # build our dependencies
        removal_service = RemovalService()
        reference = UploadService(removal_service)

        # call upload_complete, which should, in turn, call `rm`:
        reference.upload_complete("my uploaded file")
```

```
        # check that it called the rm method of any RemovalService
        mock_rm.assert_called_with("my uploaded file")
```

```
        # check that it called the rm method of _our_ removal_service
        removal_service.rm.assert_called_with("my uploaded file")
```

Ми перевірили, що клас UploadService успішно викликає метод екземпляру rm(). Зауважте, що механізм патчингу насправді замінив метод rm() у всіх екземплярах RemovalService в тестовому методі.

Особливості патчингу: порядок декораторів. При використанні кількох декораторів над тестовими методами важливий порядок! У своїй основі при відображенні декораторів на параметри методу декоратори працюють у [зворотному порядку](). Розглянемо приклад:

```
    @mock.patch('mymodule.sys')
    @mock.patch('mymodule.os')
    @mock.patch('mymodule.os.path')
    def test_something(self, mock_os_path, mock_os, mock_sys):
        pass
```

Зверніть увагу, що параметри зіставляються в оберненому порядку відносно декораторів. Часто це зумовлено особливостями роботи Python. У псевдокоді це виглядатиме приблизно так:

```
patch_sys(patch_os(patch_os_path(test_something)))
```

Оскільки патч для `sys` знаходиться ззовні, його виконання очікується останнім, тобто відповідний параметр тестового методу теж буде наприкінці.

СПОСІБ 2. СТВОРЕННЯ МОК-ЕКЗЕМПЛЯРІВ

Замість мокування конкретного методу екземпляра можна постачати тільки мокований екземпляр в UploadService за допомогою його конструктора. Спосіб 1 більш точний та конкретний, проте цей спосіб може бути в деяких ситуаціях ефективнішим. Знову здійснимо рефакторинг тесту:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from mymodule import RemovalService, UploadService
```

```
import mock
import unittest

class RemovalServiceTestCase(unittest.TestCase):
    @mock.patch('mymodule.os.path')
    @mock.patch('mymodule.os')
    def test_rm(self, mock_os, mock_path):
        # instantiate our service
        reference = RemovalService()

        # set up the mock
        mock_path.isfile.return_value = False

        reference.rm("any path")

        # test that the remove call was NOT called.
        self.assertFalse(mock_os.remove.called, "Failed to not remove the
file if not present.")

        # make the file 'exist'
        mock_path.isfile.return_value = True

        reference.rm("any path")

        mock_os.remove.assert_called_with("any path")


class UploadServiceTestCase(unittest.TestCase):
    def test_upload_complete(self, mock_rm):
        # build our dependencies
        mock_removal_service = mock.create_autospec(RemovalService)
        reference = UploadService(mock_removal_service)

        # call upload_complete, which should, in turn, call `rm`:
        reference.upload_complete("my uploaded file")

        # test that it called the rm method
```

```
        mock_removal_service.rm.assert_called_with("my uploaded file")
```

У даному прикладі не довелось навіть патчити функціональність – просто створили автоспецифікацію (`mock.create_autospec(RemovalService)`) для об'єктів класу `RemovalService`, а далі впроваджуємо (inject) отриманий об'єкт в класі UploadService, щоб перевірити (validate) функціональність. Метод mock.create_autospec() створює функціонально еквівалентний екземпляр переданого класу. На практиці це означає, що при взаємодії з поверненим об'єктом викидатимуться винятки при некоректному використанні. Конкретніше, якщо метод викликається не неправильною кількістю аргументів. Це особливо важливо в процесі рефакторингу. Під час змін у бібліотеці тест очікувано ламається. Без використання автоспецифікації тести все ще будуть проходити, незважаючи на збійність реалізації, що лежить в основі.

Бібліотека mock також включає 2 важливих класи, на базі яких побудовано більшість внутрішньої функціональності: mock.Mock та mock.MagicMock. При виборі використання екземпляру класу mock.Mock чи mock.MagicMock завжди віддавайте перевагу автоспецифікації, оскільки вона тримає тести «в тонусі» щодо майбутніх змін. Це пов'язано з тим, що ці класи приймають всі виклики методів та присвоєння значень властивостей незалежно від API, що лежить в основі. Розглянемо такий приклад:

```
class Target(object):
    def apply(value):
        return value

def method(target, value):
    return target.apply(value)
```

Відтестувати цей код за допомогою екземпляру класу mock.Mock можна так:

```
class MethodTestCase(unittest.TestCase):
    def test_method(self):
        target = mock.Mock()

        method(target, "value")
        target.apply.assert_called_with("value")
```

Дана логіка здається правильною, проте спробуйте внести зміни в метод Target.apply(), щоб він приймав більше параметрів:

```
class Target(object):
    def apply(value, are_you_sure):
        if are_you_sure:
```

```
            return value
        else:
            return None
```

Перезапустіть тест і побачите, що він все ще проходить. Це пов’язано з тим, що тест збирається не відносно поточного API! Тому *завжди* використовуйте метод create_autospec() та параметр autospec з декораторами @patch та @patch.object.

Таким чином, бібліотека mock дещо складна для роботи, проте значно вплинула на модульне тестування Python-коду. Джерело.

3. Ознайомтесь з тестуванням засобами фреймворку pytest, повторивши операції, представлені в главах 1 і 2.

ІНДЗ

1. Продовжіть виконувати завдання 3 з практичної роботи для решти глав 3-7.