# ІНФОРМАТИКА ТА ПРОГРАМУВАННЯ

## Тема 30. Тестування. Розповсюдження власних застосувань

# Тестування

- У цій темі ми трохи відійдемо від розгляду прикладних задач та їх розв’язування та подивимось на створення програм з іншого боку.
- Просто написати програму – це добре.
- Але, якщо ставити на меті створити програму, якою будуть користуватись чи яка буде вирішувати промислові задачі, треба приділити увагу тестуванню та розповсюдженню власних застосувань.
- **Тестування** (software testing) – це діяльність, виконувана для оцінки та вдосконалення програмного забезпечення.
- Ця діяльність, у загальному випадку, базується на виявленні дефектів і проблем у програмних системах.
- Розрізняють різні рівні та типи тестування: модульне, інтеграційне, функціональне, приймально-здавальне та інші.
- Ми розглянемо модульне тестування, тому що воно здійснюється розробниками програм та полягає у написанні програм, що перевіряють інші програми (модулі).

# Модульне тестування

- **Модульне тестування** (unit testing) спрямоване на перевірку функціонування окремо взятого елемента системи.
- Модульне тестування полягає у виділенні набору тестових сценаріїв (test cases), написанні програм для реалізації цих сценаріїв, виконанні написаних програм та оцінці результатів.
- Окремий тестовий сценарій перевіряє, чи видає при заданих початкових даних частина програмного коду (як правило, функція або метод) результат, який відповідає очікуваному.
- Застосування модульного тестування дозволяє автоматизувати процес тестування та виконувати його багаторазово, при змінах програми або її частини.

# Засоби для модульного тестування у Python

- У якості засобів модульного тестування у Python застосовують пакети unittest та doctest, що входять до стандартної поставки Python, а також бібліотеки від інших компаній та незалежних розробників (наприклад, nose).
- unittest використовують для побудови та виконання тестових сценаріїв.
- Окремий сценарій оформлюють у вигляді класу.
- Більш докладно цей пакет ми розглянемо пізніше.

# Засоби для модульного тестування у Python.2

- doctest шукає тести у документації (коментарях), що включена у код програми, та виконує їх, а також перевіряє результати.
- doctest вважає, що тестом є будь-який рядок, що починається з стандартної підказки інтерпретатора Python (‘>>>’), після цього рядка можуть йти рядки коду програми, що починаються з трьох крапок (‘...’).
- Вважається, що у рядку, який йде після рядків коду, вказано очікуваний результат тесту.

# Засоби для модульного тестування у Python.3

- Наприклад, якщо у модулі mymodule є функція fact, яка обчислює факторіал натурального числа, а також коментар, що містить рядок:

```
"""
>>>from mymodule import fact


>>>p = fact(3)
6
"""
```

- doctest буде вважати це тестом, виконає його та підтвердить правильність результату.

# Пакет unittest та клас TestCase

- Пакет unittest виконує модульне тестування.
- Для цього у пакет включено декілька модулів та класів.
- Ми розглянемо модулі unittest та unittest.mock.
- Одним з головних класів модуля unittest є клас TestCase.
- Цей клас призначено для створення об’єктів, що містять методи для проведення тестування.
- Щоб написати власний тестовий сценарій, потрібно описати клас-нащадок класу TestCase та включити у нього методи, що здійснюють тестування.
- Варто зазначити, що імена цих методів можуть бути довільними, але повинні починатись з рядка “test”.
- Для запуску тестів (методів), що зібрано у деякому модулі у класах-нащадках TestCase, треба у цьому модулі викликати функцію main модуля unittest:

```
unittest.main()
```

# Методи перевірки правильності тверджень

- Для перевірки правильності тверджень щодо функціонування програми, яка тестується, у класі TestCase є низка методів, імена яких починаються з “Assert”.
- Кожен з цих методів перевіряє одну умову (один тест), що стосується виконання програми.
- Якщо ця умова задовольняється, тест вважається успішним, інакше тест вважається неуспішним (fail).
- Більшість цих методів в якості одного або двох перших параметрів мають вираз(и).
- Останнім необов’язковим параметром (msg) є повідомлення, яке буде показано у разі неуспішності тесту.
- Наприклад, виклик

```
tc.AssertEqual(x, 0, "{}!= 0".format(x))
```

  - де tc – об’єкт класу TestCase, у випадку, якщо x дорівнює 5, покаже повідомлення “5 != 0”.

# Методи перевірки правильності тверджень.2

- Деякі методи перевірки правильності тверджень щодо функціонування програми зібрано у таблиці:

| Метод | Опис |
|---|---|
| tc.assertEqual(a, b, msg=None) | Перевіряє, що a == b |
| tc.assertNotEqual(a, b, msg=None) | Перевіряє, що a != b |
| tc.assertTrue(x, msg=None) | Перевіряє, що bool(x) є істиною (True) |
| tc.assertFalse(x, msg=None) | Перевіряє, що bool(x) є хибністю (False) |
| tc.assertIn(a, b) | Перевіряє, що a входить у b |
| tc.assertNotIn(a, b) | Перевіряє, що a не входить у b |
| tc.assertRaises(exc, fun, *args, **kwds, msg=None) | Функція fun(*args, **kwds) ініціює виключення exc |
| tc.assertGreater(a, b, msg=None) | Перевіряє, що a > b |
| tc.assertGreaterEqual(a, b, msg=None) | Перевіряє, що a >= b |
| tc.assertLess(a, b, msg=None) | Перевіряє, що a < b |
| tc.assertLessEqual(a, b, msg=None) | Перевіряє, що a <= b |

# Приклад: Тестування функції, яка перевіряє, чи є рядок паліндромом

- Скласти тести для функції, яка перевіряє, чи є рядок паліндромом.
- Щоб здійснити тестування функції, яка перевіряє, чи є рядок паліндромом (is_palindrome), треба розглянути, як мінімум, такі твердження:
  - порожній рядок є паліндромом
  - симетричний рядок з парною кількістю символів є паліндромом
  - симетричний рядок з непарною кількістю символів є паліндромом
  - рядок, у якому символи можуть бути у різних регістрах, але який однаково читається з початку та кінця, є паліндромом
  - рядок, у якому є символи-розділювачі, але який однаково читається з початку та кінця, є паліндромом
  - несиметричний рядок не є паліндромом

# Тестування функції перевірки, чи є рядок паліндромом. Реалізація

- Для перевірки функції is_palindrome опишемо клас TestIsPalindrome.
- У цьому класі реалізуємо методи, які перевіряють твердження, сформульовані вище:
  - test_1_isempty - порожній рядок є паліндромом
  - test_2_iseven - симетричний рядок з парною кількістю символів є паліндромом
  - test_3_isodd - симетричний рядок з непарною кількістю символів є паліндромом
  - test_4_iscase - рядок, у якому символи можуть бути у різних регістрах, але який однаково читається з початку та кінця, є паліндромом
  - test_5_isdelim - рядок, у якому є символи-розділювачі, але який однаково читається з початку та кінця, є паліндромом
  - test_6_isnot - несиметричний рядок не є паліндромом
- Версія 2 прикладу відрізняється тим, що у виклику unittest.main вказано ключовий параметр verbosity=2, що означає виведення на екран інформації навіть для тестів, які виконуються успішно.

# Підготовка та очищення тестового оточення

- Для тестування часто необхідно спеціально підготувати тестове оточення або створити так званий «випробувальний стенд» (test fixture).
- Після тестування цей випробувальний стенд слід очистити «розібрати», щоб він не впливав на подальше виконання програми.
- Методи для підготовки та очищення тестового оточення також містяться у класі TestCase.
- Це методи setUp та tearDown відповідно.
- Метод setUp викликається перед викликом кожного методу, що виконує тест, а tearDown – після виклику кожного методу, що виконує тест.

# Приклад: Тестування визначення розміру каталогів (Версія 1)

- У темі «Використання операційної системи» ми розглядали приклад програми, яка визначає розміри усіх підкаталогів заданого каталогу.
- Необхідно виконати тестування цієї програми.
- Вказана програма містить дві функції, які й треба тестувати:
  - getdirsize – повертає розмір заданого каталогу разом з усіма підкаталогами
  - getdirslist – отримує список підкаталогів заданого каталогу разом з їх розмірами. Список містить кортежі (<розмір>, <каталог>).
- Спочатку перевіримо правильність роботи функцій, якщо заданий каталог є порожнім.
- У цьому випадку getdirsize повинна повертати 0, а getdirslist – порожній список.
- Опишемо клас TestEmptyDirSize – нащадок.TestCase, який містить 2 методи для тестування згаданих функцій: test_1_dirsize_one та test_2__dirsize_all.
- Також опишемо методи для побудови та очищення тестового оточення: setUp та tearDown. setUp створює порожній каталог _test, а tearDown, - видаляє його.

# Приклад: Тестування визначення розміру каталогів (Версія 2)

- Необхідно виконати тестування програми, яка визначає розміри усіх підкаталогів заданого каталогу.
- У другій версії ми на додачу до класу TestEmptyDirSize опишемо клас TestNotEmptyDirSize, який здійснює тестування для непорожніх каталогів.
- Відповідно, треба підготувати ці непорожні каталоги та записати у них файли заданої довжини, щоб потім перевірити результат.
- Клас TestNotEmptyDirSize містить ті ж методи, що й клас TestEmptyDirSize плюс внутрішні методи _write_one_file – записати один файл заданої довжини- та _make_one_dir – створити каталог з заданою кількістю файлів.
- Метод setUp створює каталог _test, що містить 2 підкаталоги, один з яких містить підкаталог, а метод tearDown, - видаляє каталог _test разом з підкаталогами та файлами.

# Удавані об’єкти

- При тестуванні часто виникає необхідність замінити деяку існуючу функцію або модуль, клас удаваним об’єктом (mock object).
- Удавані об’єкти потрібні у випадках:
  - ізоляції програми, яка тестується
  - використання зовнішніх програмних інтерфейсів
  - відсутності реальних об’єктів
- Ізоляція програми, яка тестується, - це відключення посилань на зовнішні модулі для того, щоб перевірити тільки функціональність даного модуля.
- Використання зовнішніх програмних інтерфейсів може мати певні обмеження.
- Наприклад, обмеження за швидкістю з’єднання, або обмеження на кількість безоплатних запитів.
- У таких випадках при тестуванні використовують удавані об’єкти замість безпосереднього звернення до зовнішнього інтерфейсу.
- Відсутність реальних об’єктів можлива, якщо модуль, у якому повинні створюватись ці об’єкти ще не розроблено.
- Звичайно, інтерфейс модуля у цьому випадку вже повинен бути відомий.
- Тоді удавані об’єкти використовують замість реальних об’єктів до їх створення.

# Функція patch

- У Python заміна реального об’єкту удаваним здійснюється за допомогою функції patch.
- Функція patch реалізована у модулі unittest.mock.
- Функцію patch можна використовувати як декоратор з параметрами, як менеджер контексту або як звичайну функцію.
- Найчастіше patch використовують як декоратор.
- Цей декоратор застосовують до методу, який виконує тестування, наприклад

```
@patch("object_to_mock")
def testmethod(self, mock_object):
    ...
```

  - де object_to_mock – об’єкт, який треба замінити, mock_object – удаваний об’єкт, який буде використовуватись у методі тестування testmethod та функціях, які він викликає, замість object_to_mock.
- Треба відмітити, що застосування patch додає один додатковий параметр до методу тестування.

# Функція patch.2

- Використання patch у якості менеджеру контексту виглядає так:

```
with patch("object_to_mock") as mock_object:
    ...
```

- Після заміни реального об’єкту удаваним ми можемо вказувати значення, що повертається

```
mock_object.return_value = e
```

- або вказувати «побічний ефект» виконання функції у вигляді виключення або послідовності (для моделювання генератора)

```
mock_object.side_effect = lst
```

- або взагалі замінити цей об’єкт власною функцією (або модулем, класом)

```
mock_object = myfunc
```

# Приклад: Тестування визначення розміру каталогів з використанням patch

- Виконати тестування програми, яка визначає розміри усіх підкаталогів заданого каталогу, ізолювавши цей модуль від модуля os.
- Для ізоляції програми від модуля os застосуємо функцію patch як декоратор:

```
@patch("T22.t22_01_dirsize_v1.os")
def test_1_dirsize_one(self, my_os):
    ...
```

- Таким чином, ми замінюємо весь модуль os на удаваний об’єкт my_os.
- У методі test_1_dirsize_one необхідно задати значення, що повертаються, для усіх функцій модуля os, які ми використовуємо у функції, що тестується, або задати нові функції замість функцій модуля os (my_join, my_walk).
- Версія 2 програми відрізняється тим, що демонструє використання patch в якості менеджера контексту.
- Також замість власних іменованих функцій my_join та my_walk використовується лямбда-функція та побічний ефект.

# Розповсюдження власних застосувань

- Розповсюдження власних застосувань полягає у підготовці інсталяційного пакету та його розміщенні на доступному ресурсі.
- Цим доступним ресурсом може бути просто диск комп'ютера або ресурс у мережі.
- Як правило, у мережі використовують загальнодоступний ресурс Python Package Index або PyPi.
- Щоб розмістити свій пакет на цьому ресурсі, треба створити на ньому аккаунт та зареєструвати свій проект.
- Спочатку рекомендується попрактикуватись на тестовому ресурсі https://testpypi.python.org/pypi.
- Є різниця у підготовці пакетів, що містять тільки Python, та пакетів, які мають вставки з бінарним кодом (частіше – скомпільовані файли C).
- Так само, є особливості у розповсюдженні пакетів, що залежать від тієї чи іншої операційної системи.
- Ми розглянемо найпростіший варіант підготовки до розповсюдження пакету, який містить тільки програму у Python.

# Підготовка структури каталогів для власного пакету

- Спочатку треба створити порожній каталог, ім’я якого співпадає з ім’ям пакету, що розповсюджується.
- Потім – наповнити його підкаталогами та файлами.
- Цей каталог повинен мати приблизно таку структуру:

```
packagename/
    README.txt
    doc/
        documentation.txt
    packagename/
        __init__.py
        module_1.py
        …
        module_n.py
    utils/
        __init__.py
        util.py
    examples/
        example_1.py
```

# Підготовка структури каталогів для власного пакету.2

- Якщо пакет супроводжується тестами, то додається підкаталог tests.
- Можуть бути додані й інші підкаталоги, або забрані зображені (наприклад, utils)
- Файл README.txt містить короткий опис пакету та його застосування, а файл documentation.txt – більш докладний опис.
- Файли __init__.py можуть містити описи глобальних констант або залишатись порожніми

# Написання setup.py та файлу MANIFEST.in

- Після створення та наповнення каталогу пакету необхідно створити та зберегти у каталозі packagename/ файл setup.py.
- Цей файл має приблизно такий вигляд:

```
# setup.py
from distutils.core import setup

setup(name='packagename',
      version=<версія>,
      author=<автор>,
      author_email=<пошта автора>,
      url=<url автора>,
      packages=['packagename', 'packagename.utils']
      )
```

- Функція setup здійснює підготовку пакету до розповсюдження. distutils.core – це модуль, який містить функцію setup.

# Написання setup.py та файлу MANIFEST.in.2

- Окрім файлу setup.py, у каталозі packagename/ треба створити текстовий файл MANIFEST.in, у якому вказати файли, що не є модулями пакету, але потрібні для його розповсюдження.
- Наприклад, файл

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include doc *
```

- вказує на те, що у пакет необхідно включити усі текстові файли з каталогу packagename/ а також усі файли з каталогів packagename/doc та packagename/examples.

# Створення архіву та подальша інсталяція пакету

- Створення архіву, що буде містити інсталяцію пакету здійснюють командою ОС:

  python setup.py sdist

- Результатом виконання цієї команди буде створення каталогу packagename/dist та запис у нього заархівованого файлу, який містить усе необхідне для подальшої інсталяції пакету.
- Для подальшої інсталяції пакет можна розмістити на PyPi або залишити на диску.
- Інсталяція здійснюється за допомогою стандартної програми pip.
- Якщо пакет завантажено на PyPi, достатньо виконати команду ОС

  pip install *packagename*

- Для інсталяції з диску потрібно вказати повний шлях до каталогу пакету pakagename_path

  pip install *packagename_path*

# Приклад: Розповсюдження пакету для резервного копіювання (backup)

- Створити файл для розповсюдження пакету, що здійснює резервне копіювання файлів з заданих каталогів.
- Саму програму ми розглядали у темі «Використання операційної системи».
- Щоб побудувати пакет для подальшого розповсюдження, створимо каталог demobackup (так буде називатись наш пакет) разом з потрібними підкаталогами:

```
demobackup/
    README.txt
    doc/
        documentation.txt
    demobackup/
        __init__.py
        backup.py
        config.txt
        config_dict.py
        schedule.py
    examples/
        backup_example.py
```

# Приклад: Розповсюдження пакету для резервного копіювання (backup).2

- Створимо та запишемо у каталог demobackup/ файли setup.py:

```
# setup.py
from distutils.core import setup

setup(name='demobackup',
      version='1.0',
      author='Alexander Obvintsev',
      author_email='obvintsev.stud@gmail.com',
      url='http://matfiz.univ.kiev.ua/',
      packages=['demobackup'],
      )
```

- та MANIFEST.in:

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include doc *
recursive-include demobackup *.txt
```

# Приклад: Розповсюдження пакету для резервного копіювання (backup).3

- Після цього командою ОС

  python setup.py sdist

- побудуємо каталог dist та заархівований файл demobackup-1.0.zip.
- Для подальшої інсталяції з диску потрібно вказати

  pip install *path*

- де path – шлях до каталогу demobackup.

# Резюме

- Ми розглянули:
  1. Тестування. Модульне тестування
  2. Засоби для модульного тестування у Python
  3. Пакет unittest та клас TestCase
  4. Методи перевірки правильності тверджень
  5. Підготовка та очищення тестового оточення
  6. Удавані об’єкти. Функція patch
  7. Розповсюдження власних застосувань
  8. Підготовка структури каталогів для власного пакету
  9. Написання setup.py та файлу MANIFEST.in
  10. Створення архіву та подальша інсталяція пакету

# Де прочитати

1. Peter Norton, Alex Samuel, David Aitel та інші - Beginning Python
2. David Beazley - Python Cookbook, 3rd edition – 2013
3. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
4. Magnus Lie Hetland - Beginning Python from Novice to Professional, 2nd ed – 2008
5. Python 3.4.3 documentation
6. https://blog.fugue.co/2016-02-11-python-mocking-101.html