GeekBrains

Лекция 14. Основы тестирования

Погружение в Python

Оглавление

На этой лекции мы

1. Разберёмся с написанием тестов в Python
2. Изучим возможности doctest
3. Узнаем о пакете для тестирования unittest
4. Разберёмся с тестированием через pytest

Дополнительные материалы к лекции

1. Библиотека mock объектов https://docs.python.org/3.11/library/unittest.mock.html
2. Полная документация по pytest https://docs.pytest.org/en/stable/contents.html

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с обработкой ошибок в Python
2. Изучили иерархию встроенных исключений
3. Узнали о способе принудительного поднятия исключения в коде
4. Разобрались в создании собственных исключений

Термины лекции

- **Тестирование программного обеспечения** — это исследование, проводимое с целью предоставления заинтересованным сторонам информации о качестве тестируемого программного продукта или услуги. Тестирование программного обеспечения также может обеспечить объективный, независимый взгляд на программное обеспечение, позволяющий бизнесу оценить и понять риски внедрения программного обеспечения.
- **Простое число** — натуральное число, имеющее ровно два различных натуральных делителя. Другими словами, натуральное число p является простым, если оно отлично от 1 и делится без остатка только на 1 и на само p.
- **Натуральные числа (от лат. naturalis «естественный»)** — числа, возникающие естественным образом при счёте (1, 2, 3, 4, 5, 6, 7 и так далее).
- **Разработка через тестирование (англ. test-driven development, TDD)** — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Подробный текст лекции

0. Основы тестирования

Тестирование кода является неотъемлемой частью больших проектов. Мало кто пишет тесты для одноразовых задач в несколько строчек кода. Но если ваш проект планирует развиваться, тесты помогут экономить время при расширении функционала, внесении доработок в существующий код.

Обычно тестирование подразделяется на три категории:

1. Функциональное тестирование
    - Модульное (компонентное)
    - Интеграционное
    - Системное

    - Регрессионное
    - Приемочное
    - Смоук
2. Тестирование производительности
    - Тестирование отказоустойчивости
    - Нагрузочное
    - Объемное
    - Тестирование масштабируемости
3. Обслуживание (регресс и обслуживание)
    - Регрессионное
    - Тестирование технического обслуживания

Список далеко не полный. Да и способов группировки тестирования больше много больше. В рамках данного курса и лекции не будут рассматриваться все возможные виды тестирования. Главная цель занятия - познакомить с тремя основными инструментами, позволяющими писать тесты для ваших Python проектов. При этом в рамках проекта обычно используется один из трёх рассматриваемых вариантов, а не все разом. По традиции разберём возможные варианты на примерах кода с пояснениями.

1. Основы doctest

Инструмент doctest встроен в Python и не требует дополнительных манипуляций по установке и настройке. Как заявляют сами разработчики языка doctest можно использовать для следующих задач:

- Проверка актуальности строк документации модуля путем проверки того, что все интерактивные примеры по-прежнему работают в соответствии с документацией.
- Для регрессионного тестирования. Чтобы убедиться, что интерактивные примеры из тестового файла или тестового объекта работают должным образом.
- Позволяют написать учебную документацию для пакета, обильно иллюстрированную примерами ввода-вывода. В зависимости от того, что выделено — примеры или пояснительный текст, это может быть что-то вроде “грамотного тестирования” или “исполняемой документации”.

Проверка примеров в документации, регрессионное тестирование

Как вы помните из прошлых лекций, тройные двойные кавычки сразу после заголовка класса, функции или метода превращают текст внутри в строку документации соответствующего объекта. Например так может выглядеть простейшая (без оптимизации) функция, проверяющая является ли число простым или составным используя нахождение остатка от деления.

```
def is_prime(p: int) -> bool:
    """
       Checks the number P for simplicity using finding the
remainder of the division in the range [2, P).
    """
    for i in range(2, p):
        if p % i == 0:
            return False
    return True


help(is_prime)
```

А теперь сохраним код в файле main.py и сделаем несколько запусков в терминале в режиме интерпретатора.

```
>>> from main import is_prime
Help on function is_prime in module main:
is_prime(p: int) -> bool
       Checks the number P for simplicity using finding the
remainder of the division in the range [2, P).
>>> is_prime(42)
False
>>> is_prime(73)
True
```

1. Мы сразу вспомнили, что команда import запускает импортируемый файл. У нас сработал вызов справки, потому что мы не спрятали его в __name__ == “__main__”
2. Вызов функции в режиме интерпретатора позволяет получить ответ для любых значений.

Если перенести вызову и результаты из консоли в строку документации функции (класса, модуля), получим тесты doctest. В нашем случае можно сделать так:

```
def is_prime(p: int) -> bool:
```

```
    """
       Checks the number P for simplicity using finding the
remainder of the division in the range [2, P).
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
    """
    for i in range(2, p):
        if p % i == 0:
            return False
    return True


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

В документацию добавлены строки вызова функции в режиме интерпретатора. Они начинаются с тройной стрелки и пробела. Сразу после идёт строка с ответом.
В “нейм-мейн” спрятали импорт модуля doctest и вызов функции testmod для тестирования кода.

🔥 **Важно!** doctest запускает код и сравнивает возвращаемое значение в виде текста с текстов внутри строки документации. Если допустить опечатку, поставить лишние отступы или ещё как-то изменить текст ответа, тест будет провален.

При запуске файла ничего не происходит. По умолчанию тестирование не выводит информации, если тесты прошли успешно. Попросим добавить вывод результатов в любом случае. Для этого исправим последнюю строку на `doctest.testmod(verbose=True)`
Теперь перед нами подробный вывод того, что тесты пройдены успешно.

Разработка через тестирование, TDD

Разработка через тестирование (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое

изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.
В TDD выделяют следующие этапы:

1. Добавление теста
2. Запуск всех тестов: убедиться, что новые тесты не проходят
3. Написать код
4. Запуск всех тестов: убедиться, что все тесты проходят
5. Рефакторинг
6. Повторить цикл

Применим TDD на практике.

➢ **Добавление теста**

Попробуем добавить в нашу функцию тесты для проверки особых случаев, а именно:

- Число должно быть натуральным.
    - Если функция вызывается не с целым, будем возвращать ошибку типа.
    - А если число будет целым, но не натуральным, ошибку значения.
- Отдельно напишем тест для единицы - натурального целого числа, которое не может быть проверено на простоту.
- Предусмотреть предупреждение о возможно долгом поиске ответа, если на простоту проверяется число больше ста миллионов.
    - Сделаем тест для большого составного числа
    - Отдельно сделаем тест для большого простого числа

В результате написания тестов получим следующий код:

```
def is_prime(p: int) -> bool:
    """
       Checks the number P for simplicity using finding the
remainder of the division in the range [2, P).
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
    >>> is_prime(3.14)
    Traceback (most recent call last):
        ...
    TypeError: The number P must be an integer type
    >>> is_prime(-100)
    Traceback (most recent call last):
    ...
    ValueError: The number P must be greater than 1
```

```
    >>> is_prime(1)
    Traceback (most recent call last):
    ...
    ValueError: The number P must be greater than 1
    >>> is_prime(100_000_001)
     If the number P is prime, the check may take a long time.
Working...
    False
    >>> is_prime(100_000_007)
     If the number P is prime, the check may take a long time.
Working...
    True
    """
    for i in range(2, p):
        if p % i == 0:
            return False
    return True


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Обратите внимание на многоточия в тестах ошибок. Модуль doctest ориентируется на первую строку об ошибке. Это или Traceback (most recent call last): или Traceback (innermost last): Так как номера строк кода, пути имена могут меняться, середина вывода игнорируется. Её можно заменить на многоточие. Важна последняя строка, которая начинается с типа ошибки и последующего за ней текста.

➢ **Запуск всех тестов: убедиться, что новые тесты не проходят**

Запускаем файл с кодом даже не включая режим отображения verbose. И получаем огромный список ошибок заканчивающийся следующим итогом:

```
**********************************************************************
*****
1 items had failures:
   5 of   7 in __main__.is_prime
***Test Failed*** 5 failures.
```

Логично провалить 5 новых тестов, ведь мы пока не писали код, который позволит их пройти.

🔥 **Важно!** Так как doctest сравнивает текст, вызов ошибки, например через raise и печать аналогичного текста через print() будут восприниматься одинаково.

➢ **Написать код**

Самое время написать несколько строчек кода внутри функции is_prime(). На выходе получим следующий файл:

```
def is_prime(p: int) -> bool:
    """
       Checks the number P for simplicity using finding the
remainder of the division in the range [2, P).
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
    >>> is_prime(3.14)
    Traceback (most recent call last):
        ...
    TypeError: The number P must be an integer type
    >>> is_prime(-100)
    Traceback (most recent call last):
    ...
    ValueError: The number P must be greater than 1
    >>> is_prime(1)
    Traceback (most recent call last):
    ...
    ValueError: The number P must be greater than 1
    >>> is_prime(100_000_001)
     If the number P is prime, the check may take a long time.
Working...
    False
    >>> is_prime(100_000_007)
     If the number P is prime, the check may take a long time.
Working...
    True
    """
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than 1')
    elif p > 100_000_000:
          print('If the number P is prime, the check may take a
long time. Working...')
    for i in range(2, p):
```

```
        if p % i == 0:
            return False
    return True


if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Понадобилось написать шесть строк кода, три проверки.

1. Убедиться что число целое, иначе вызвать ошибку типа.
2. Убедиться, что число не меньше двух, иначе вызвать ошибку значения. Тут мы проходим сразу два теста из пяти добавленных в техзадании.
3. Убедиться, что число меньше ста миллионов, иначе вывести предупреждение. Снова закрываем два теста одной проверкой

➢ **Запуск всех тестов: убедиться, что все тесты проходят**

Запуск кода без режима verbose ничего не выводит. Код успешно проходит тесты. С режимом отображения увидим успешное прохождение всех семи тестов:

```
...
7 passed and 0 failed.
Test passed.
```

Ура! Мы успешно справились с поставленной задачей.

Проверка исполняемой документации

Проект по созданию модуля работы с простыми числами оказался успешным. Мы сохранили код в prime.py и активного расширяем его возможности. Самое время написать файл документации к модулю. Создаём prime.md — текстовый файл, поддерживающий разметку Markdown.

```
Документация к модулю работы с простыми числами
===

Описание функции is_prime()
---

Для проверки числа на простоту используйте функцию is_prime
```

```
модуля prime.
Импортируйте её в свой код.

    >>> from prime import is_prime

Теперь можно проверять числа на простоту.

    >>> is_prime(2)
    True

Функция использует проверку остатка от деления и может долго
возвращать результат для больших простых чисел.
Вы получите предупреждение, это нормально. Просто подождите.

    >>> is_prime(100000007)
      If the number P is prime, the check may take a long time.
Working...
    True

*Приятного использования нашего модуля ;-)*
```

🔥 **Важно!** Обычно документацию пишут на английском языке. В учебном примере специально был выбран русский как основной язык на учебной платформе. Вы всегда можете использовать современные online переводчики для получения нужного языка.

В документации есть несколько примеров, имитирующих работу в режиме интерпретатора. Убедимся, что они рабочие, написав в отдельном файле пару строк кода.

```
import doctest

doctest.testfile('prime.md', verbose=True)
```

Функция testfile построчно читает переданный ей файл и если встречает примеры выполнения кода, тестирует их работоспособность. Обратите внимание, что мы указываем на необходимость импорта функции в самом начале документации. И после вызова оставили пустую строку. doctest сделает импорт и не будет ожидать ничего в ответ. Да, это строчка будет воспринята как тест. И без него все последующие примеры провалятся. Ведь режим интерпретатора работает последовательно.

Запуск тестов из командной строки

Прежде чем завершить введение работу с модулем doctest пара примеров запуска из терминала.

🔥 **Важно!** Не путайте терминал (консоль, командную строку) операционной системы и консольный режим работы интерпретатора Python.

```
PS C:\Users\PycharmProjects> python -m doctest .\prime.py
PS C:\Users\PycharmProjects> python -m doctest .\prime.py -v
PS C:\Users\PycharmProjects> python -m doctest .\prime.md
PS C:\Users\PycharmProjects> python -m doctest .\prime.md -v
```

Вызываем интерпретатор python и в качестве модуля указываем doctest. Далее передаём путь до файла, который хотим тестировать. Если файл имеет расширение py, запускаеты функция testmod (строки 1 и 2). А если у файла другое расширение, предполагается что это исполняемая документация и запускается функция testfile (строки 3 и 4). Дополнительный ключ -v включает режим подробного вывода результатов тестирования.

🔥 **Внимание!** Пример кода сделан в терминале PowerShell. В зависимости от используемого вами терминала строка приглашения может быть другой. Но текст команд одинаков для любой ОС и любого терминала.

Задание

Перед вами несколько строк doctest. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
"""
>>> say('Hello')
Hello Hello
>>> say('Hi', 5)
Hi Hi Hi Hi Hi
>>> say('cat', 3, '(=^.^=)')
cat(=^.^=)cat(=^.^=)cat
"""
```

2. Основы тестирования с unittest

Рассмотрим более мощный по функциональности инструмент тестирования из коробки. Модуль unittest входит в стандартную библиотеку Python и не требует дополнительной установки. Более того, unittest называют фреймворком, а не просто модулем.
Среда unittest модульного тестирования изначально была вдохновлена JUnit и имеет тот же вкус, что и основные среды модульного тестирования на других языках. Он поддерживает автоматизацию тестирования, совместное использование кода установки и завершения тестов, объединение тестов в коллекции и независимость тестов от структуры отчетности.

🔥 **Внимание!** Вдохновение JUnit сказалось на стиле фреймворка, а именно на использовании camelCase для имён, вместо привычного для Python разработчика стиля snake_case.

Общие моменты работы с unittest

Рассмотрим некоторые общие моменты работы с unittest на примере следующего кода.

```
import unittest


class TestCaseName(unittest.TestCase):

    def test_method(self):
        self.assertEqual(2 * 2, 5, msg='Видимо и в этой вселенной не работает :-(')


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

Для хранения тестов рекомендуется создавать отдельный файл с тестами или папку tests, если файлов с тестами будет много. Смешивать в одном файле исполняемый код и тесты не рекомендуется.
В файле с тестом импортируем модуль unittest и создаём класс для тестирования - test case. Такой класс должен наследоваться от TestCase.
Внутри класса создаём методы, имена которых должны начинаться со слова test. Таких методов внутри класса может быть несколько.
По наследованию от класса TestCase и именам методов unittest понимает, что перед ним тесты, которые необходимо запустить.
Для проверки используем утверждения - “ассерты”. В приведённом примере assertEqual принимает два аргумента: 2 * 2 и 5. Тест утверждает, что они равны. А если значения не равны, будет поднято исключение AssertionError с текстом, который передали в ключевом параметре msg.

🔥 **Внимание!** Реальные тесты не должны содержать неверные утверждения, подобные “дважды два равно пяти”.

Для запуска тестов вызываем функцию main(). Она проанализирует файл, соберёт тестовые кейсы, запустит и сообщит результаты проверки.

🔥 **Внимание!** Команда для запуска тестов из командной строки выглядит аналогично запуску doctest

```
$ python3 -m unittest tests.py -v
```

Сравнение тестов doctest и unittest

Возьмём уже знакомую функцию проверки числа на простоту и реализуем написанные ранее в doctest тесты используя unittest.

Файл prime.py без тестов doctest

```
def is_prime(p: int) -> bool:
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than one')
    elif p > 100_000_000:
          print('If the number P is prime, the check may take a
```

```
long time. Working...')
    for i in range(2, p):
        if p % i == 0:
            return False
    return True
```

Ничего нового в коде функции нет.

А так будет выглядет файл test_prime.py

```
import io
import unittest
from unittest.mock import patch

from prime import is_prime


class TestPrime(unittest.TestCase):

    def test_is_prime(self):
        self.assertFalse(is_prime(42))
        self.assertTrue(is_prime(73))

    def test_type(self):
        self.assertRaises(TypeError, is_prime, 3.14)

    def test_value(self):
        with self.assertRaises(ValueError):
            is_prime(-100)
            is_prime(1)

    @patch('sys.stdout', new_callable=io.StringIO)
    def test_warning_false(self, mock_stdout):
        self.assertFalse(is_prime(100_000_001))
        self.assertEqual(mock_stdout.getvalue(),
                         'If the number P is prime, the check may
take a long time. Working...\n')

    @patch('sys.stdout', new_callable=io.StringIO)
    def test_warning_true(self, mock_stdout):
        self.assertTrue(is_prime(100_000_007))
        self.assertEqual(mock_stdout.getvalue(),
                         'If the number P is prime, the check may
take a long time. Working...\n')


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

Разберём каждый из тестов внутри класса:

➢ **Кейс test_is_prime**

Проверяем базовую работу функции. Утверждение assertFalse ожидает получить ложь в качестве аргумента. В нашем случае в качестве результата вызова функции. Аналогично assertTrue ожидает получить истину.

➢ **Кейс test_type**

Утверждение assertRaises ожидает ошибку типа (аргумент один) если вызвать функцию is_prime (аргумент два) и передать ей число 3.14 (аргумент три).

➢ **Кейс test_value**

Используем менеджер контекста для утверждения ошибки и внутри контекста дважды запускаем функцию. assertRaises во всех случаях будет ожидать ошибку значения

➢ **Кейсы test_warning_false и test_warning_true**

🔥 **Внимание!** Оба примера выходят за рамки основ unittest. Это скорее пример на будущее для самых любознательных.

Используя декоратор patch из модуля mock перенаправляем стандартный поток вывода sys.stdout обращаясь к StringIO модуля ввода-вывода io. Результат попадает в параметр mock_stdout. Внутри метода делаем стандартную проверку на ложь или истину для большого числа. А далее проверяем, что стандартный вывод получил значение, совпадающее с ожидаемым текстом предупреждения.

🔥 **Внимание!** Разбор Mock объектов выходит за рамки лекции. Самые любознательные могут обратиться к стандартной документации языка. https://docs.python.org/3.11/library/unittest.mock.html

Запуск тестов doctest из unittest

А что если тесты уже написаны в doctest? В этом случае можно создать функцию test_loader и добавить тесты doctest в перечень для тестирования. Изучите пример.

```
import doctest
import unittest

import prime


def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(prime))
    tests.addTests(doctest.DocFileSuite('prime.md'))
    return tests


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

Объект tests используя метод addTests добавляет импортированный модуль prime. Для этого используется класс DocTestSuite из модуля doctest. А если необходимо тестировать документацию, используется класс DocFileSuite. Теперь функция unittest.main соберёт написанные раннее тесты doctest и запустит их.

Подготовка теста и сворачивание работ

Иногда бывает необходимо выполнить какие-то действия до начала тестирования, развернуть тестируемую среду. А после завершения теста наоборот, убрать лишнее. Для этих целей в unittest есть зарезервированные имена методов setUp и tearDown. Часто их называют фикстурами.

- ➢ **Метод setUp**

Когда внутри класса есть несколько тестовых методов, вызов метода setUp происходит перед каждым вызовом теста.

```
import unittest


class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        self.data = [2, 3, 5, 7]
        print('Выполнил setUp')  # Только для демонстрации работы
метода

    def test_append(self):
```

```
        self.data.append(11)
        self.assertEqual(self.data, [2, 3, 5, 7, 11])

    def test_remove(self):
        self.data.remove(5)
        self.assertEqual(self.data, [2, 3, 7])

    def test_pop(self):
        self.data.pop()
        self.assertEqual(self.data, [2, 3, 5])


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

В примере трижды создаётся список на четыре элемента. Каждый из тестов ожидает, что будет работать с числами 2, 3, 5, 7 и никак не учитывает результаты работы других тестов. Подобный подход удобен, когда надо прогнать большое количество тестов на одном и том же наборе данных.

➢ **Метод tearDown**

Метод tearDown будет вызван после успешного выполнения метода setUp и в случае если тест отработал успешно, и если он провалился.

```
import unittest


class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        with open('top_secret.txt', 'w', encoding='utf-8') as f:
            for i in range(10):
                f.write(f'{i:05}\n')

    def test_line(self):
        with open('top_secret.txt', 'r', encoding='utf-8') as f:
            for i, line in enumerate(f, start=1):
                pass
        self.assertEqual(i, 10)

    def test_first(self):
        with open('top_secret.txt', 'r', encoding='utf-8') as f:
            first = f.read(5)
            self.assertEqual(first, '00000')

    def tearDown(self) -> None:
```

```
        from pathlib import Path
        Path('top_secret.txt').unlink()


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

В примере метод setUp создаёт перед каждым тестом файл со строками чисел. Два теста работают с этим файлом. И после каждого происходит удаление файла из tearDown метода.
Даже если провалить тест, файл будет удалён.

Перечень доступных утверждений assert

В списке ниже приведены доступные в unittest утверждения и пояснения о том что именно они проверяют.

- assertEqual(a, b) - a == b
- assertNotEqual(a, b) - a != b
- assertTrue(x) - bool(x) is True
- assertFalse(x) - bool(x) is False
- assertIs(a, b) - a is b
- assertIsNot(a, b) - a is not b
- assertIsNone(x) - x is None
- assertIsNotNone(x) - x is not None
- assertIn(a, b) - a in b
- assertNotIn(a, b) - a not in b
- assertIsInstance(a, b) - isinstance(a, b)
- assertNotIsInstance(a, b) - not isinstance(a, b)
- assertRaises(exc, fun, *args, **kwds) - функция fun(*args, **kwds) поднимает исключение exc
- assertRaisesRegex(exc, r, fun, *args, **kwds) - функция fun(*args, **kwds) поднимает исключение exc и сообщение совпадает с регулярным выражением r
- assertWarns(warn, fun, *args, **kwds) - функция fun(*args, **kwds) поднимает предупреждение warn
- assertWarnsRegex(warn, r, fun, *args, **kwds) - функция fun(*args, **kwds) поднимает предупреждение warn и сообщение совпадает с регулярным выражением r
- assertLogs(logger, level) - блок with записывает логи в logger с уровнем level

- assertNoLogs(logger, level) - блок with не записывает логи в logger с уровнем level
- assertAlmostEqual(a, b) - round(a-b, 7) == 0
- assertNotAlmostEqual(a, b) - round(a-b, 7) != 0
- assertGreater(a, b) - a > b
- assertGreaterEqual(a, b) - a >= b
- assertLess(a, b) - a < b
- assertLessEqual(a, b) - a <= b
- assertRegex(s, r) - r.search(s)
- assertNotRegex(s, r) - not r.search(s)
- assertCountEqual(a, b) - a и b содержат одни и те же элементы в одинаковом количестве независимо от их порядка в коллекциях

Как вы видите перечень допустимых проверок достаточно обширный, чтобы удовлетворить практически любые запросы по написанию тестов.

Задание

Перед вами несколько строк кода. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
import unittest
from main import func


class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        self.data = {'one': 1, 'two': 2, 'three': 3, 'four': 4}

    def test_step_1(self):
        self.assertEqual(func(self.data), 4)

    def test_step_2(self):
        self.assertEqual(func(self.data, first=False), 2)


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

3. Основы тестирования с pytest

Финальный модуль для создания тестов — pytest. Он не входит в стандартную библиотеку Python, поэтому должен быть установлен перед использованием.

```
pip install pytest
```

🔥 **Важно!** Версия Python должна быть 3.7 или выше.

Команда assert

В Python есть зарезервированное слово assert. После работы с unittest вы догадываетесь о её назначении. assert делает утверждение. Если оно возвращает истину, программа продолжает работать. А если утверждение ложно, поднимается ошибка AssertionError.
Для простоты можно представить assert как особую конструкцию if.

- “асерт”

```
assert утверждение, "Утверждение не подтвердилось"
```

- “иф”

```
if утверждение:
    pass
else:
    raise AssertionError("Утверждение не подтвердилось")
```

Модуль pytest формирует свою работу вокруг встроенной команды assert. Но в отличии от примера с if даёт подробную информацию об ошибках, если тесты не проходят.

Общие моменты работы с pytest

Рассмотрим простой пример тестирования работы функции, которая складывает два числа.

```
import pytest


def sum_two_num(a, b):
    return a + b
    # return f'{a}{b}'


def test_sum():
    assert sum_two_num(2, 3) == 5, 'Математика покинула чат'


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

Импорт модуля нужен только для запуска тестов из файла. Для создания простейших тестов модуль pytest не нужен.
Функция sum_two_num наш подопытный. Она принимает пару числе и возвращает их сумму.
Для создания кейса просто определяем функцию, которая начинается со слова test. Внутри используем assert для проверки утверждения. В простейшем случае это сравнение вызова функции с ожидаемым результатом. Более сложные ассерты рассмотрим далее. Дополнительно можно указать сообщение, которое будет выведено в случае провала теста.
Количество функций в файле может быть любым. pytest найдёт и запустит их все на основе сопоставления имён. При этом превращать функции в методы класса как в unittest не нужно. А если очень хочется объединить кейсы внутри класса, создайте класс начинающийся с Test.
Чтобы запустить тест из файла, вызываем функцию main из модуля pytest.

🔥 **Внимание!** Команда для запуска тестов из командной строки выглядит аналогично запуску doctest и unittest. Ключ с одиночным или двойным v указывает на уровень детализации. Кроме того можно вызывать pytest напрямую.

```
$ python3 -m pytest tests.py -vv
$ pytest tests_pt.py
```

Сравнение тестов pytest с doctest и unittest

Ещё раз возьмём функцию проверки числа на простоту и реализуем написанные ранее тесты используя pytest.

Файл prime.py не изменился

```
def is_prime(p: int) -> bool:
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than one')
    elif p > 100_000_000:
          print('If the number P is prime, the check may take a
long time. Working...')
    for i in range(2, p):
        if p % i == 0:
            return False
    return True
```

Файл test_prime_pt.py с кейсами pytest

```
import pytest

from prime import is_prime


def test_is_prime():
    assert not is_prime(42), '42 - составное число'
    assert is_prime(73), '73 - простое число'


def test_type():
    with pytest.raises(TypeError):
        is_prime(3.14)


def test_value():
    with pytest.raises(ValueError):
        is_prime(-100)


def test_value_with_text():
```

```
    with pytest.raises(ValueError, match=r'The number P must be
greater than 1'):
        is_prime(1)


def test_warning_false(capfd):
    is_prime(100_000_001)
    captured = capfd.readouterr()
    assert captured.out == 'If the number P is prime, the check
may take a long time. Working...\n'


def test_warning_true(capfd):
    is_prime(100_000_007)
    captured = capfd.readouterr()
    assert captured.out == 'If the number P is prime, the check
may take a long time. Working...\n'


if __name__ == '__main__':
    pytest.main(['-v'])
```

Начало с импортом и конец с запуском тестов стандартные для Python. Разберём каждый из кейсов.

➢ **Кейс test_is_prime**

Проверяем базовую работу функции. Утверждение assert not ожидает получить ложь в качестве результата вызова функции. Второй assert ожидает получить истину.

🔥 **Важно!** Если первая строка провалит тест, второй assert не будет вызван для проверки. Обычно внутри кейса пишут одно утверждения. В нашем случае можно разделить проверки на два отдельных кейса.

➢ **Кейс test_type**

Используем менеджер контекста pytest.raises который ожидает получить ошибку TypeError при вызове is_prime с вещественным число в качестве аргумента. Наличие ошибки проходит тест, а её отсутствие - роняет. Строка, которая должна поднять ошибку - последняя строка внутри менеджера контекста. Дальнейший код не будет выполняться.

➢ **Кейс test_value**

Тест работает аналогично проверки типа, но мы указали другую ошибку в менеджере и передали другое значение в функцию.

➢ **Кейс test_value_with_text**

Более сложный подход к тестированию. Помимо ошибки в параметр match передаётся регулярное выражение. Если оно совпадёт с текстом ошибки, тест будет пройден.

➢ **Кейсы test_warning_false и test_warning_true**

Внимание! Оба примера выходят за рамки основ pytest. Это скорее пример на будущее для самых любознательных.
Кейс получает фикстуру capfd в качестве аргумента. capfd (capture file descriptors) является одной из встроенных в pytest фикстур, которая позволяет перехватывать потоки вывода и ошибок. Внутри кейса вызываем тестируемую функцию. Далее используем метод readouterr() для получение потоков в переменную captured. В финале сравниваем результат captured.out (потока вывода) с ожидаемым текстом сообщения.

Запуск тестов doctest и unittest

Для запуска тестов, написанных другими инструментами можно воспользоваться следующими командами в консоли ОС:

```
$ pytest --doctest-modules prime.py -v
$ pytest tests_ut.py
```

В случае с doctest необходимо указать флаг --doctest-modules. Для unittest ничего указывать не надо. Практически все кейс из unittest можно запускать в pytest. Модуль понимает синтаксис, способен собрать тесты из классов и проверить их.

Фикстуры pytest как замены unittest setUp и tearDown

Если вам необходимо выполнить однотипные действия для подготовки нескольких тестов, можно создать собственные фикстуры. Рассмотрим простой пример из главы о unittest.

```
import pytest


@pytest.fixture
def data():
    return [2, 3, 5, 7]


def test_append(data):
    data.append(11)
    assert data == [2, 3, 5, 7, 11]


def test_remove(data):
    data.remove(5)
    assert data == [2, 3, 7]


def test_pop(data):
    data.pop()
    assert data == [2, 3, 5]


if __name__ == '__main__':
    pytest.main(['-v'])
```

Функция data превращается в фикстуру добавлением декоратора @pytest.fixture. Чтобы использовать фикстуру внутри кейса, необходимо передать её в качестве аргумента. Выбранный разработчика pythest подход к фикстурам удобен тем, что позволяет любые вариации с кейсами.

- можно иметь множество фикстур и разные кейсы могут использовать разные фикстуры.
- в кейс можно передать любое количество фикстур.
- фикстура может принимать в качестве аргумента другую фикстуру.

Ещё немного о фикстурах

Рассмотрим пример посложнее.

```
import pytest


@pytest.fixture
def get_file(tmp_path):
    f_name = tmp_path / 'test_file.txt'
    print(f'Создаю файл {f_name}')  # принтим в учебных целях
    with open(f_name, 'w+', encoding='utf-8') as f:
        yield f
    print(f'Закрываю файл {f_name}')  # принтим в учебных целях


@pytest.fixture
def set_num(get_file):
    print(f'Заполняю файл {get_file.name} цифрами')  # принтим в
учебных целях
    for i in range(10):
        get_file.write(f'{i:05}')
    get_file.seek(0)


@pytest.fixture
def set_char(get_file):
    print(f'Заполняю файл {get_file.name} буквами')  # принтим в
учебных целях
    for i in range(65, 91):
        get_file.write(f'{chr(i)}')
    get_file.seek(0)
    return get_file


def test_first_num(get_file, set_num):
    first = get_file.read(5)
    assert first == '00000'


def test_first_char(set_char):
    first = set_char.read(5)
    assert first == 'ABCD'  # специально провалим тест


if __name__ == '__main__':
    pytest.main(['-v'])
```

Кейсов всего два: test_first_num и test_first_chr. И каждый из них использует свои фикстуры. Но давайте обо всём сверху вниз.

- **Фикстура get_file**

Фикстура принимает на вход аргумент tmp_path. Это встроенная фикстура, которая возвращает временный путь - объект pathlib.Path. При использовании выводим не печать информацию о создании файла и о его удалении. Отследим когда срабатывает фикстура.
Внутри менеджера контекста создаём файл и через команду yield возвращаем указатель на него. Если бы мы использовали команду return, менеджер контекста вызвал бы f.close() после возврата указания и файл стал бы нечитаемым.
Используя yield мы превратили функцию в генератор. Теперь внутри фикстуры есть “сетап” создающий файл и “тирдаун”, закрывающий его после использования.

Внимание! Мы явно не удаляем временные файлы. Фикстура tmp_path сохраняет три последних временных каталога, удаляя старые при очередном запуске.

- **Фикстура set_num**

Используя файловый дескриптор get_file записываем строку из цифр и возвращаем указатель на начало файла. Фикстура ничего не возвращает.

- **Фикстура set_char**

Снова используем файловый дескриптор get_file, но получаем уже другой файл. Имя совпадает, но каталоги разные. Заполняем его буквами, сбрасываем позицию в ноль и возвращаем get_file - файловый дескриптор.

- **Кейс test_first_num**

Перед началом теста срабатывают фикстуры, создающие временный файл и заполняющие его цифрами. Далее обращаемся к get_file чтобы прочитать пять первых символов и сравниваем их со строкой текста.

- **Кейс test_first_char**

Кейс получает всего одну фикстуру set_char. Но так как она самостоятельно вызывает фикстуру get_file и возвращает её, мы можем обращаться к файловому дескриптору по имени set_char.

Рассмотренный пример даёт представление о гибкости кейсов pytest.
Субъективное мнение автора курса, но pytest является лучшим из трёх рассмотренных инструментов тестирования. Попробуйте все три, составьте своё.

Задание

Перед вами несколько строк кода. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
import pytest
from main import func


def test_1():
    assert func(4) == 0


def test_2():
    assert func(4, -4) == (1, 0)


def test_3():
    assert func(4, -10, -50) == (5, -2.5)


def test_4():
    assert func(1, 1, 1) is None


if __name__ == '__main__':
    pytest.main(['-v'])
```

Вывод

На этой лекции мы:

1. Разобрались с написанием тестов в Python
2. Изучили возможности doctest
3. Узнали о пакете для тестирования unittest
4. Разобрались с тестированием через pytest

Краткий анонс следующей лекции

1. Узнаем о составе стандартной библиотеки Python
2. Разберёмся в настройках логирования
3. Изучим работу с датой и временем
4. Узнаем ещё пару полезных структур данных
5. Изучим способы парсинга аргументов при запуске скрипта с параметрами

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий. Напишите к ним тесты. Попробуйте написать одинаковые тесты в трёх инструментах. Так у вас будет возможность сравнить их.