10-单元测试

- 什么是单元测试
- 安装和配置测试环境
- 测试函数
- 测试类
- 测试模拟对象

安装和配置测试环境

- 安装 pytest： pip install pytest
- 在vscode中安装 Python Test Explorer for Visual Studio Code 插件
- 在vscode中配置python的单元测试
  - 在vscode左侧点击Testing
  - 在测试面板点击按钮Configure Python Test
  - 选择pytest测试框架，就能配置完成
  - 配置好测试框架后，可以直接在TEST EXPLORER中查看和运行单元测试

测试函数

被测试的函数文件名： name_function.py

In [ ]:
```
def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

手动输入测试

In [ ]:
```
from name_function import get_formatted_name


print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

自动的单元测试相比手动测试：

- 测试速度更快，效率更高
- 单元测试是可以被复用的

- 软件质量更有保障

使用单元测试框架进行

- 使用不同的参数作为函数的输入参数调用函数，每一组输入称作一个测试用例
- 获取到每个测试用例输入时函数的返回结果
- 使用 `assert` 断言语句判断，函数的返回结果是否和测试用例的预期结果一致

创建函数的单元测试

- 创建文件 `test_name_function.py`
- 导入被测试的模块的函数
- 创建测试方法以 `test_` 开头
- 运行测试：
  - 在终端运行pytest命令
  - 在测试面板运行测试

In [ ]:
```
from name_function import get_formatted_name


def test_first_last_name():
    """Do names like 'Janis Joplin' work?"""
    formatted_name = get_formatted_name('janis', 'joplin')
    assert formatted_name == 'Janis Joplin'

def test_first_last_middle_name():
    """Do names like 'Wolfgang Amadeus Mozart' work?"""
    formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
    assert formatted_name == 'Wolfgang Amadeus Mozart'
```

测试类

文件名： `survey.py`

In [ ]:
```python
class AnonymousSurvey:
    """Collect anonymous answers to a survey question."""

    def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

    def show_question(self):
        """Show the survey question."""
        print(self.question)

    def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

    def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

常用的一些断言语句

Assertion	Claim
assert a == b	断言两个值相等
assert a != b	断言两个值不相等
assert a	断言a为True
assert not a	断言a为False
assert element in list	断言一个元素在列表中
assert element not list	断言一个元素不在列表中

对类进行单元测试：

- 导入pytest和要测试的类
- 使用 `@pytest.fixture` 创建可重复使用的测试装置
- 创建测试方法
- 运行测试

In [ ]:
```python
import pytest
from survey import AnonymousSurvey


@pytest.fixture
def language_survey():
    """A survey that will be available to all test functions."""
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
    return language_survey

def test_store_single_response(language_survey):
    """Test that a single response is stored properly."""
    language_survey.store_response('English')
    assert 'English' in language_survey.responses

def test_store_three_responses(language_survey):
    """Test that three individual responses are stored properly."""
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        language_survey.store_response(response)

    for response in responses:
        assert response in language_survey.responses
```

利用模拟（mock）对象测试

模拟对象（mock object）用于在测试环境中替换和模拟真实对象。

模拟的对象：

- 日期数据
- 访问某个外部服务
- 文件数据
- 系统的API
- 用户的输入和事件
- ……

为什么要使用模拟对象？

测试的成功以否应该不依赖外部条件，模拟对象可以使得测试不再依赖这些外部条件。

例如：你的代码需要使用Http请求访问外部服务，你的测试代码需要执行这个请求去访问外部服务是否符合你的预期需求。 但是，有时外部服务可能会间断地失败。

这时可以使用模拟对象替换真正的Http请求，让你模拟访问外部服务来测试你的代码是否符合预期。

利用 pytest-mock 模块来创建模拟对象

- 安装 pytest-mock 模块： pip install pytest-mock
- 查看文档pytest-mock doc (https://pytest-mock.readthedocs.io/en/latest)

In [ ]:
```
import os

class UnixFS:

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

In [ ]:
```
def test_unix_fs(mocker):
    mocker.patch('os.remove')
    UnixFS.rm('file')
    os.remove.assert_called_once_with('file')
```