Lecture 5 第 5 讲

Unit Tests 单元测试

- Up until now, you have been likely testing your own code using `print` statements.
  到目前为止，您可能一直在使用 `print` 语句测试自己的代码。
- Alternatively, you may have been relying upon CS50 to test your code for you!
  或者，您可能一直依赖 CS50 来为您测试代码！
- It's most common in industry to write code to test your own programs.
  在行业中，编写代码来测试您自己的程序是最常见的。
- In your console window, type `code calculator.py`. Note that you may have previously coded this file in a previous lecture. In the text editor, make sure that your code appears as follows:
  在控制台窗口中，键入 `code calculator.py`。请注意，您可能之前在之前的讲座中对此文件进行了编码。在文本编辑器中，确保您的代码如下所示：

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

  Notice that you could plausibly test the above code on your own using some obvious numbers such as `2`. However, consider why you might want to create a test that ensures that the above code functions appropriately.
  请注意，您可以使用一些明显的数字（如 `2`）自行合理地测试上述代码。但是，请考虑为什么您可能希望创建一个测试来确保上述代码正常运行。
- Following convention, let's create a new test program by typing `code test_calculator.py` and modify your code in the text editor as follows:
  按照惯例，让我们通过键入 `代码来` 创建新的测试程序test_calculator.py并在文本编辑器中修改您的代码，如下所示：

```
from calculator import square


def main():
    test_square()


def test_square():
    if square(2) != 4:
        print("2 squared was not 4")
    if square(3) != 9:
        print("3 squared was not 9")


if __name__ == "__main__":
    main()
```

  Notice that we are importing the `square` function from `square.py` on the first line of code. By convention, we are creating a function called `test_square`. Inside that function, we define some conditions to test.
  请注意，我们在第一行代码中从 `square.py` 导入 `square` 函数。按照惯例，我们将创建一个名为 `test_square` 的函数。在该函数中，我们定义了一些要测试的条件。
- In the console window, type `python test_calculator.py`. You'll notice that nothing is being outputted. It could be that everything is running fine! Alternatively, it could be that our test function did not discover one of the "corner cases" that could produce an error.
  在控制台窗口中，键入 `python test_calculator.py`。您会注意到没有输出任何内容。可能是一切运行正常！或者，可能是我们的测试函数没有发现可能产生错误的 "极端情况" 之一。
- Right now, our code tests two conditions. If we wanted to test many more conditions, our test code could easily become bloated. How could we expand our test capabilities without expanding our test code?
  现在，我们的代码测试两个条件。如果我们想测试更多的条件，我们的测试代码很容易变得臃肿。我们如何在不扩展测试代码的情况下扩展测试功能？

`assert` 断言

- Python's `assert` command allows us to tell the compiler that something, some assertion, is true. We can apply this to our test code as follows:
  Python 的 `assert` 命令允许我们告诉编译器某些东西，某个断言，是真的。我们可以将其应用于我们的测试代码，如下所示：

```
from calculator import square


def main():
    test_square()
```

```
def test_square():
    assert square(2) == 4
    assert square(3) == 9


if __name__ == "__main__":
    main()
```

Notice that we are definitively asserting what `square(2)` and `square(3)` should equal. Our code is reduced from four test lines down to two.

请注意，我们明确地断言 `square（2）` 和 `square（3）` 应该等于什么。我们的代码从 4 个测试行减少到 2 个。

- We can purposely break our calculator code by modifying it as follows:

  我们可以通过修改 calculator 代码来故意破坏它，如下所示：

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n + n


if __name__ == "__main__":
    main()
```

Notice that we have changed the `*` operator to a `+` in the square function.

请注意，我们已将 square 函数中的 `*` 运算符更改为 `+`。

- Now running `python test_square.py` in the console window, you will notice that an `AssertionError` is raised by the compiler. Essentially, this is the compiler telling us that one of our conditions was not met.

  现在，在控制台窗口中运行 `python test_square.py`，您会注意到编译器引发了 `AssertionError`。实质上，这是编译器告诉我们未满足我们的一个条件。

- One of the challenges that we are now facing is that our code could become even more burdensome if we wanted to provide more descriptive error output to our users. Plausibly, we could code as follows:

  我们现在面临的挑战之一是，如果我们想为用户提供更具描述性的错误输出，我们的代码可能会变得更加繁琐。合理地，我们可以按如下方式编码：

```
from calculator import square


def main():
    test_square()


def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared is not 4")
    try:
        assert square(3) == 9
    except AssertionError:
        print("3 squared is not 9")
    try:
        assert square(-2) == 4
    except AssertionError:
        print("-2 squared is not 4")
    try:
        assert square(-3) == 9
    except AssertionError:
        print("-3 squared is not 9")
    try:
        assert square(0) == 0
    except AssertionError:
        print("0 squared is not 0")


if __name__ == "__main__":
    main()
```

Notice that running this code will produce multiple errors. However, it's not producing all the errors above. This is a good illustration that it's worth testing multiple cases such that you might catch situations where there are coding mistakes.

请注意，运行此代码将产生多个错误。但是，它不会产生上述所有错误。这是一个很好的例子，表明值得测试多种情况，这样你就可以发现存在编码错误的情况。

- The above code illustrates a major challenge: How could we make it easier to test your code without dozens of lines of code like the above?

  上面的代码说明了一个重大挑战：我们如何才能在没有像上面这样的几十行代码的情况下更轻松地测试您的代码？

You can learn more in Python's documentation of `assert`.

您可以在 Python 的 `assert` 文档中了解更多信息。

`pytest`

- `pytest` is a third-party library that allows you to unit test your program. That is, you can test your functions within your program.

  `PyTest` 是一个第三方库，允许您对程序进行单元测试。也就是说，您可以在程序中测试您的函数。

- To utilize `pytest` please type `pip install pytest` into your console window.

  要使用 `pytest`，请在控制台窗口中键入 `pip install pytest`。

- Before applying `pytest` to our own program, modify your `test_calculator` function as follows:

  在将 `pytest` 应用到我们自己的程序之前，请修改你的 `test_calculator` 函数，如下所示：

```
from calculator import square


def test_assert():
```

```
def test_assert():
    assert square(2) == 4
    assert square(3) == 9
    assert square(-2) == 4
    assert square(-3) == 9
    assert square(0) == 0
```

Notice how the above code asserts all the conditions that we want to test.

请注意上面的代码如何断言我们要测试的所有条件。

- `pytest` allows us to run our program directly through it, such that we can more easily view the results of our test conditions.

  `pytest` 允许我们直接通过它运行我们的程序，这样我们就可以更容易地查看测试条件的结果。

- In the terminal window, type `pytest test_calculator.py`. You'll immediately notice that output will be provided. Notice the red `F` near the top of the output, indicating that something in your code failed. Further, notice that the red `E` provides some hints about the errors in your `calculator.py` program. Based upon the output, you can imagine a scenario where `3 * 3` has outputted `6` instead of `9`. Based on the results of this test, we can go correct our `calculator.py` code as follows:

  在终端窗口中，键入 `pytest test_calculator.py`。您会立即注意到将提供 output。请注意输出顶部附近的红色 `F`，表示代码中的某些内容失败。此外，请注意，红色 `E` 提供了一些有关 `calculator.py` 程序中错误的提示。根据输出，您可以想象 `3 * 3` 输出 `6` 而不是 `9` 的场景。根据这个测试的结果，我们可以按照如下方式更正我们的 `calculator.py` 代码：

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

Notice that we have changed the `+` operator to a `*` in the square function, returning it to a working state.

请注意，我们已在 square 函数中将 `+` 运算符更改为 `*`，使其返回到工作状态。

- Re-running `pytest test_calculator.py`, notice how no errors are produced. Congratulations!

  重新运行 `pytest test_calculator.py`，请注意如何不产生错误。祝贺！

- At the moment, it is not ideal that `pytest` will stop running after the first failed test. Again, let's return our `calculator.py` code back to its broken state:

  目前，`pytest` 在第一次测试失败后停止运行并不理想。同样，让我们将 `calculator.py` 代码返回到其损坏状态：

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n + n


if __name__ == "__main__":
    main()
```

Notice that we have changed the `*` operator to a `+` in the square function, returning it to a broken state.

请注意，我们已在 square 函数中将 `*` 运算符更改为 `+`，将其返回到损坏状态。

- To improve our test code, let's modify `test_calculator.py` to divide the code into different groups of tests:

  为了改进我们的测试代码，让我们修改 `test_calculator.py` 以将代码划分为不同的测试组：

```
from calculator import square


def test_positive():
    assert square(2) == 4
    assert square(3) == 9


def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9


def test_zero():
    assert square(0) == 0
```

Notice that we have divided the same five tests into three different functions. Testing frameworks like `pytest` will run each function, even if there was a failure in one of them. Re-running `pytest test_calculator.py`, you will notice that many more errors are being displayed. More error output allows you to further explore what might be producing the problems within your code.

请注意，我们已将相同的 5 个测试划分为 3 个不同的函数。像 `pytest` 这样的测试框架将运行每个函数，即使其中一个函数出现故障。重新运行 `pytest test_calculator.py`，你会注意到显示更多错误。通过更多错误输出，您可以进一步探索可能在代码中产生问题的原因。

- Having improved our test code, return your `calculator.py` code to fully working order:

  改进我们的测试代码后，将您的 `calculator.py` 代码恢复到完全正常工作的顺序：

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

Notice that we have changed the `+` operator to a `*` in the square function, returning it to a working state.

请注意，我们已在 square 函数中将 `+` 运算符更改为 `*`，使其返回到工作状态。

- Re-running `pytest test_calculator.py`, you will notice that no errors are found.

  重新运行 `pytest test_calculator.py`，您会注意到没有找到错误。

- Finally, we can test that our program handles exceptions. Let's modify `test_calculator.py` to do just that.

  最后，我们可以测试我们的程序是否处理异常。让我们修改 `test_calculator.py` 来做到这一点。

```
import pytest

from calculator import square


def test_positive():
    assert square(2) == 4
    assert square(3) == 9


def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9


def test_zero():
    assert square(0) == 0


def test_str():
    with pytest.raises(TypeError):
        square("cat")
```

Notice that instead of using `assert`, we are taking advantage of a function within the `pytest` library itself called `raises` which allows you to express that you expect an error to be raised. We need to go to the top of our program and add `import pytest` and then call `pytest.raises` with the type of error we are expecting.

请注意，我们没有使用 `assert`，而是利用了 `pytest` 库本身中一个名为 `raises` 的函数，它允许你表示你希望引发错误。我们需要转到程序的顶部并添加 `import pytest`，然后使用我们预期的错误类型调用 `pytest.raises`。

- Again, re-running `pytest test_calculator.py`, you will notice that no errors are found.

  同样，重新运行 `pytest test_calculator.py`，您会注意到没有发现任何错误。

- In summary, it's up to you as a coder to define as many test conditions as you see fit!

  总之，作为编码人员，您可以定义您认为合适的尽可能多的测试条件！

You can learn more in Pytest's documentation of `pytest`.

您可以在 Pytest 的 `pytest` 文档中了解更多信息。

Testing Strings 测试字符串

- Going back in time, consider the following code `hello.py`:

  回到过去，请考虑以下代码 `hello.py`：

  ```
  def main():
      name = input("What's your name? ")
      hello(name)


  def hello(to="world"):
      print("hello,", to)


  if __name__ == "__main__":
      main()
  ```

  Notice that we may wish to test the result of the `hello` function.

  请注意，我们可能希望测试 `hello` 函数的结果。

- Consider the following code for `test_hello.py`:

  请考虑以下代码进行 `test_hello.py`：

  ```
  from hello import hello


  def test_hello():
      assert hello("David") == "hello, David"
      assert hello() == "hello, world"
  ```

  Looking at this code, do you think that this approach to testing will work well? Why might this test not work well? Notice that the `hello` function in `hello.py` prints something: That is, it does not return a value!

  查看此代码，您认为这种测试方法是否有效？为什么这项测试效果不佳？请注意，`hello.py` 中的 `hello` 函数会打印一些内容：也就是说，它不会返回值！

- We can change our `hello` function within `hello.py` as follows:

  我们可以在 `hello.py` 中更改我们的 `hello` 函数，如下所示：

  ```
  def main():
      name = input("What's your name? ")
      print(hello(name))


  def hello(to="world"):
      return f"hello, {to}"


  if __name__ == "__main__":
      main()
  ```

Notice that we changed our `hello` function to return a string. This effectively means that we can now use `pytest` to test the `hello` function.

请注意，我们更改了 `hello` 函数以返回一个字符串。这实际上意味着我们现在可以使用 `pytest` 来测试 `hello` 函数。

- Running `pytest test_hello.py`, our code will pass all tests!

  运行 `pytest test_hello.py`，我们的代码将通过所有测试！

- As with our previous test case in this lesson, we can break out our tests separately:

  与本课前面的测试用例一样，我们可以单独分解测试：

```
from hello import hello


def test_default():
    assert hello() == "hello, world"


def test_argument():
    assert hello("David") == "hello, David"
```

Notice that the above code separates our test into multiple functions such that they will all run, even if an error is produced.

请注意，上面的代码将我们的测试分成多个函数，这样即使产生错误，它们也会全部运行。

Organizing Tests into Folders
将测试组织到文件夹中

- Unit testing code using multiple tests is so common that you have the ability to run a whole folder of tests with a single command.

  使用多个测试的单元测试代码非常常见，以至于您可以使用单个命令运行整个测试文件夹。

- First, in the terminal window, execute `mkdir test` to create a folder called `test`.

  首先，在终端窗口中，执行 `mkdir test` 以创建一个名为 `test` 的文件夹。

- Then, to create a test within that folder, type in the terminal window `code test/test_hello.py`. Notice that `test/` instructs the terminal to create `test_hello.py` in the folder called `test`.

  然后，要在该文件夹中创建测试，请在终端窗口 `code test/test_hello.py` 中键入。请注意，`test/` 指示终端在名为 `test` 的文件夹中创建 `test_hello.py`。

- In the text editor window, modify the file to include the following code:

  在文本编辑器窗口中，修改文件以包含以下代码：

```
from hello import hello


def test_default():
    assert hello() == "hello, world"


def test_argument():
    assert hello("David") == "hello, David"
```

Notice that we are creating a test just as we did before.

请注意，我们正在像以前一样创建一个测试。

- `pytest` will not allow us to run tests as a folder simply with this file (or a whole set of files) alone without a special `__init__` file. In your terminal window, create this file by typing `code test/__init__.py`. Note the `test/` as before, as well as the double underscores on either side of `init`. Even leaving this `__init__.py` file empty, `pytest` is informed that the whole folder containing `__init__.py` has tests that can be run.

  `pytest` 不允许我们简单地使用这个文件（或一整套文件）将测试作为一个文件夹运行，而没有特殊的 `__init__` 文件。在终端窗口中，通过键入 `code test/__init__.py` 创建此文件。像以前一样注意 `test/`，以及 `init` 两侧的双下划线。即使将此 `__init__.py` 文件留空，`pytest` 也会被告知包含 `__init__.py` 的整个文件夹都有可以运行的测试。

- Now, typing `pytest test` in the terminal, you can run the entire `test` folder of code.

  现在，在终端中键入 `pytest test`，您可以运行代码的整个 `test` 文件夹。

You can learn more in Pytest's documentation of import mechanisms.

您可以在 Pytest 的导入机制文档中了解更多信息。

Summing Up 总结

Testing your code is a natural part of the programming process. Unit tests allow you to test specific aspects of your code. You can create your own programs that test your code. Alternatively, you can utilize frameworks like `pytest` to run your unit tests for you. In this lecture, you learned about...

测试代码是编程过程的自然组成部分。单元测试允许您测试代码的特定方面。您可以创建自己的程序来测试您的代码。或者，您可以利用 `pytest` 等框架为您运行单元测试。在本次讲座中，您了解了...

- Unit tests 单元测试
- `assert` 断言
- `pytest`