COMP1531

🐶 Software Engineering
1.4 - Testing - Intro

In this lecture

Why?

- Writing tests are critical to ensure an application works
- Approaching testing the right way will yield better results

What?

- Abstraction & Black boxes
- Design by contract
- Pytest

Seeing if your code works

If I left you alone right now, how would you check if this function works correctly?

```
1 def get_even(nums):
2     evens = []
3     for number in nums:
4         if number % 2 == 0:
5         evens.append(number)
6     return evens
```

even_testing.py

Seeing if your code works

Would you do something like this?

```
1  def get_even(nums):
2      evens = []
3      for number in nums:
4          if number % 2 == 0:
5          evens.append(number)
6      return evens
7
8
9  print(get_event([1,2,3]))
10 print(get_event([4,5,6]))
11 print(get_event([7]))
```

even_testing.py

Seeing if your code works

Or something like this?

```
1  def get_even(nums):
2      evens = []
3      for number in nums:
4          if number % 2 == 0:
5          evens.append(number)
6      return evens
7
8  if get_event([1,2,3]) != [2]:
9    print("Doesn't work 1")
10 if get_event([4,5,6]) != [4,6]:
11   print("Doesn't work 2")
12 if get_event([7]) != []:
13   print("Doesn't work 3")
```

even_testing.py

Using python assert

Printing errors or visually inspecting output is a method of **debugging** not **testing**. You can't call something a testing method if it doesn't **scale** well.If I left you alone right now, how would you check if this function works correctly?

Python has an **assert** function built-in that will cause an error if what it's provided is not true.

```
1  def get_evens(nums):
2      evens = []
3      for number in nums:
4          if number % 2 == 0:
5              evens.append(number)
6      return evens
7
8  assert(get_evens([1,2,3]) == [2])
9  assert(get_evens([4,5,6]) == [4,6])
10 assert(get_evens([7]) == [])
```

even_assert.py

Blackbox Testing & Abstraction

Abstraction is the notion of focusing on a higher level understanding of the problem and not worrying about the underlying detail.

We do this all the time when we drive a car, use our computers, order something online. You're typically focused on expressing an input and wanting an output, with little regard for how you get that output.

When we look at systems in an **abstract** way we could also say that we're treating them like **black boxes**.

Blackbox Testing & Abstraction

When we're **testing** our code, we always want to view the functions we're testing as abstractions / black boxes.

Let's try and write some tests

```
1 # Returns a new string with vowels removed
2 def remove_vowels(string):
3         pass
4
5 # Calculates the factorial of a number
6 def factorial(num):
7         pass
```

blackbox.py

Blackbox Testing & Abstraction

What do we notice when writing these tests?

- The tests are complete, even if they aren't being passed
- We don't need to know how the function is implemented to test the function
- Now we can go and implement it, and we have tests already done!

```
1  # Returns a new string with vowels removed
2  def remove_vowels(string):
3          pass
4
5  # Calculates the factorial of a number
6  def factorial(num):
7          pass
8
9  assert(remove_vowels("abcde") == "bcd")
10 assert(remove_vowels("frog") == "frg")
11 assert(factorial(3) == 6)
12 assert(factorial(5) == 120)
```

blackbox_testing.py

Design by contract

When we're testing or implementing a function, we will typically be working with information that tells us the constraints placed on at least the inputs.

The documentation can come in a variety of forms.

This information tells us what we do and don't need to worry about when writing tests.

```
# Returns a new string with vowels removed
# Input is a non-empty string type
# Return type is another string
def remove_vowels(string):
        pass

# Calculates the factorial of a number
# Input is a number between 1 and 10
# Output is a positive number
def factorial(num):
        pass
```

blackbox_contract.py

Systematic Python Testing

Let's take a look at **pytest**

What is pytest?

- A structured method of writing, organising, and running tests
- **pytest** is a library that helps us write small tests, but can also be used to write larger and more complex tests
- **pytest** comes with a binary that we run on command line
- **pytest** detects any **function** prefixed with **test** and runs that function, processing the assertions inside

pytest - basic

test1_nopytest.py

```
1 def sum(x, y):
2     return x * y
3
4 def test_sum1():
5     assert sum(1, 2) == 3
6
7 test_sum1()
```

```
1 $ python3 test1_nopytest.py
```

test1_pytest.py

```
1 def sum(x, y):
2     return x * y
3
4 def test_sum1():
5     assert sum(1, 2) == 3, "1 + 2 == 3"
```

```
1 $ pytest test1_pytest.py
```

pytest - more complicated

A more complicated test

test_multiple.py

```
1  import pytest
2
3  def sum(x, y):
4      return x + y
5
6  def test_small():
7      assert sum(1, 2) == 3, "1, 2 == "
8      assert sum(3, 5) == 8, "3, 5 == "
9      assert sum(4, 9) == 13, "4, 9 == "
10
11 def test_small_negative():
12     assert sum(-1, -2) == -3, "-1, -2 == "
13     assert sum(-3, -5) == -8, "-3, -5 == "
14     assert sum(-4, -9) == -13, "-4, -9 == "
15
16 def test_large():
17     assert sum(84*52, 99*76) == 84*52 + 99*76, "84*52, 99*76 == "
18     assert sum(23*98, 68*63) == 23*98 + 68*63, "23*98, 68*63 == "
```

pytest - prefixes

If you just run

$ pytest

It will automatically look for any files in that directory in the shape:

- test_*.py
- *_test.py

And then any functions that are prefixed with **test_** in those files will be run

pytest - particular files

You can run specific functions within your test files with the **-k** command. For example, we if want to run the following:

- **test_small**
- **test_small_negative**
- ~~test_large~~

We could run

$ pytest -k small

or try

$ pytest -k small -v

pytest - markers

We can also use a range of **decorators** to specify tests in python:

```python
import pytest

def pointchange(point, change):
        x, y = point
        x += change
        y += change
        return (x, y)

@pytest.fixture
def supply_point():
        return (1, 2)

@pytest.mark.up
def test_1(supply_point):
        assert pointchange(supply_point, 1) == (2, 3)

@pytest.mark.up
def test_2(supply_point):
        assert pointchange(supply_point, 5) == (6, 7)
```

```python
@pytest.mark.up
def test_3(supply_point):
        assert pointchange(supply_point, 100) == (101, 102)

@pytest.mark.down
def test_4(supply_point):
        assert pointchange(supply_point, -5) == (-4, -3)

@pytest.mark.skip
def test_5(supply_point):
        assert False == True, "This test is skipped"

@pytest.mark.xfail
def test_6(supply_point):
        assert False == True, "This test's output is muted"
```

pytest - more

There are a number of tutorials online for pytest.
This is a very straightforward one.

pytest - project structure

Whilst importing is covered in week 2, it's worth mentioning at a high level now for the project.

For the major project, your tests and implementation will be separated in different files. So writing tests will consist of importing other files to use them.

mymath.py

```
1 def sum(x, y):
2     return x * y
```

mymath_test.py

```
1 import mymath
2
3 def test_sum1():
4     assert mymath.sum(1, 2) == 3, "1 + 2 == 3"
```

```
$ pytest
```

Feedback