

# COMP1531

## 1.5 - SDLC Testing - Intro

# In this lecture

- Basics of pytest (to test code)
- Understanding importing and paths

# C-Style Testing

How did you test in COMP1511?

ctest.c

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 double sum(double a, double b) {
5     return a + b;
6 }
7
8 int main() {
9     assert(sum(1, 2) == 3);
10    assert(sum(2, 2) == 4);
11    assert(sum(3, 2) == 5);
12    printf("All tests passed\n");
13 }
```

# C-Style Testing

Let's first look at python functions

```
1 double sum(double a, double b) {  
2     return a + b;  
3 }
```

```
1 def sum(a, b):  
2     return a + b
```

Q. What are the key differences?

# C-Style Testing

Let's first look at python functions

```
1 double sum(double a, double b) {  
2     return a + b;  
3 }
```

```
1 def sum(a, b):  
2     return a + b
```

Q. What are the key differences?

- No semi-colons
- No braces
- No typing
- "def" to say define function

# C-Style Testing

Q. How would we test this python function?

```
1 def sum(a, b):  
2     return a + b
```

# C-Style Testing

Q. How would we test this python function?

cstyletest.c

```
1 def sum(a, b):  
2     return a + b  
3  
4 assert sum(1, 4) == 3
```

```
:~/teaching/cs1531/19T3-lectures/week1$ python3 cstyletest.py  
Traceback (most recent call last):  
  File "cstyletest.py", line 4, in <module>  
    assert sum(1, 2) == 3  
AssertionError
```

# C-Style Testing

Let's clean this up and wrap it in a function, though!

```
1 def sum(a, b):  
2     return a + b  
3  
4 def testSmallNumbers():  
5     assert sum(1, 4) == 3  
6  
7 testSmallNumbers()
```



# Basic Python testing

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

## What is pytest?

- 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 import pytest  
2  
3 def sum(x, y):  
4     return x * y  
5  
6 def test_sum1():  
7     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**

Without any files, it will automatically look for any files in that directory in shape:

- test\_\*.py
- \*\_test.py

# pytest - particular files

You can run specific functions without 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:

```
1 import pytest
2
3 def pointchange(point, change):
4     x, y = point
5     x += change
6     y += change
7     return (x, y)
8
9 @pytest.fixture
10 def supply_point():
11     return (1, 2)
12
13 @pytest.mark.up
14 def test_1(supply_point):
15     assert pointchange(supply_point, 1) == (2, 3)
16
17 @pytest.mark.up
18 def test_2(supply_point):
19     assert pointchange(supply_point, 5) == (6, 7)
```

```
1 @pytest.mark.up
2 def test_3(supply_point):
3     assert pointchange(supply_point, 100) == (101, 102)
4
5 @pytest.mark.down
6 def test_4(supply_point):
7     assert pointchange(supply_point, -5) == (-4, -3)
8
9 @pytest.mark.skip
10 def test_5(supply_point):
11     assert False == True, "This test is skipped"
12
13 @pytest.mark.xfail
14 def test_6(supply_point):
15     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.