# CS50P W5 - Unit Tests

1. **calculator.py**

def main():

x = int(input("What's x? "))

print("x squared is", square(x))

def square(n):

return n \* n

if \_\_name\_\_ == "\_\_main\_\_":

main()

\*Note that the if\_\_name\_\_ == “”\_\_main\_\_””: statement should be used to call main() in every program from now on.

1. Create a **test program** in a separate file : **test\_calculator.py**

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 we are importing the square() function from calculator.py. The convention is to create a function called test\_square(). Inside that function we define conditions to test.

\*This method can be limited by “corner cases” and is lengthy, we have written more code to test than to write the program.

## assert

Python’s **assert** command allows us to tell the compiler that something, some assertion, is True.



from calculator import square

def main():

test\_square()

def test\_square():

assert square(2) == 4

assert square(3) == 9

if \_\_name\_\_ == "\_\_main\_\_":

main()

\*Using the assert keyword will reduce code lines. If the assertion is True, nothing will happen. If the assertion is False, we will get an AssertionError.

1. Adding **try and except** :

from calculator import square

def main():

test\_square()

def test\_square():

try:

assert square(2) == 4

except AssertionError:

print(“2 squared was not 4”)

try:

assert square(3) == 9

except AssertionError:

print(“3 squared was not 9”)

try:

assert square(**-2**) == 4

except AssertionError:

print(“**-**2 squared was not 4”)

try:

assert square(**-3**) == 9

except AssertionError:

print(“**-**3 squared was not 9”)

try:

assert square(**0**) == 0

except AssertionError:

print(“0 squared was not 0”)

\*Adding try-except blocks will give a more user friendly output but comes with a major challenge. If we want to test for all possible cases we will need to cover all corner cases and end up with dozens of try-except blocks.

## pytest

**pytest** is a third-party library that allows you to **unit test** your program. `pip install pytest`. docs.pytest.org

from calculator import square

def test\_square():

assert square(2) == 4

assert square(3) == 9

assert square(-2) == 4

assert square(-3) == 9

assert square(0) == 0

\*pytest allows us to run our program directly through it from the terminal window : `**pytest test\_calculator.py**`

**Inconclusive:**

**```**

$ pytest test\_calculator.py

test\_calculator.py **F**  [100%]

==================== FAILURES =======================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_square \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_square():

assert square(2) == 4

> assert square(3) == 9

E assert 6 == 9

E + where 6 = square(3)

test\_calculator.py:5: AssertionError

================= short test summary info ==================

FAILED test\_calculator.py::test\_square - assert 6 == 9

================== 1 failed in 0.16s =======================

**```**

**Conclusive:**

**```**

test\_calculator.py **.**  [100%]

====================== 1 passed in 0.01s ===================

**```**

1. **Dividing tests** into groups :

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

\*It is a good idea to divide the tests into groups in different functions. pytest will run each function automatically. This will return a diagnostic of each function allowing us to have more detailed clues on the bugs.

**```**

test\_calculator.py **...**  [100%]

====================== 3 passed in 0.01s ===================

**```**

1. Handling **exceptions** :

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”)

\*We import the pytest library to use the pytest.raises() function, which allows us to express that we are expecting an error to be raised.

## Testing strings

def main():

name = input(“What’s your name? “)

hello(name)

def hello(to=”world”):

print(“hello,”, to)

if \_\_name\_\_ == "\_\_main\_\_":

main()

1. **test\_hello.py**

from hello import hello

def test\_hello():

hello("Lazer") == "hello, Lazer"

\*Attempting to use this approach to test the code might not work because the hello() function does not return a value, it simply prints something.



def main():

name = input(“What’s your name? “)

print(hello(name))

def hello(to=”world”):

return f”hello, {to}”

if \_\_name\_\_ == "\_\_main\_\_":

main()

\*Notice how we changed hello() to return instead of print. And changed the main() function to handle the printing This will allow us to use pytest. We also changed the main() function to print(hello(name)).

1. **test\_hello.py**

from hello import hello

def test\_hello():

hello("Lazer") == "hello, Lazer"

\*Now that hello() returns a value, running pytest test\_hello.py will work as expected.

1. **Dividing** tests into groups :

from hello import hello

def test\_default():

assert hello() == "hello, world"

def test\_argument():

assert hello(“Lazer”) == "hello, Lazer"

\*Dividing the test into different functions will allow us to test more effectively for each use case.

\*Suppose we have a list of names. We can test as follows:

for name in [“Abraham”, “Yitshak”, “Yaacov”]:

assert hello(name) == f”hello, {name}”

## Organizing Test into Folders

**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.

1. `**mkdir** test`
2. `**code** test**/**test\_hello.py`
3. In **test\_hello.py** write test code
4. `code test/**\_\_init\_\_.py**`

\*Used to indicate that a directory should be treated as a package, facilitating organization, importation and discovery of test modules. The file can be empty.

## convert.py

**def** main():

while True:

au = input("AU: ")

try:

au = float(au)

break

except ValueError:

continue

print(f"**{**au**}** AU is **{**convert(au)**}** m")

**def** convert(au):

if not isinstance(au, (**int**, **float**)):

raise TypeError("AU must be an int or float")

return au \* 149597870700

if \_\_name\_\_ == "\_\_main\_\_":

main()

\*isinstance(au, (int, float)) returns True if `au` is an integer or a loathing-point number. If not, the function raises a `TypeError`.

1. **test\_convert.py**

import **pytest**

from convert import convert

**def** test\_conversion():

assert convert(**1**) == 149597870700

assert convert(**50**) == 7479893535000

**def** test\_error():

with pytest.raises(TypeError):

convert(“1”)

\*pytest.raises() takes the type exception as an argument and asserts that it is raised within the with block. convert(“1”) is the statement within the `with` block and takes the string “1” as an argument.

\*Since the convert() function checks if input is int or float and raises a TypeError if not, passing string “1” will raise the exception.

1. Adjusting **float** tolerance

import **pytest**

from convert import convert

**def** test\_int\_conversion():

assert convert(**1**) == 149597870700

assert convert(**50**) == 7479893535000

**def** test\_error():

with pytest.raises(TypeError):

convert(“1”)

**def** test\_float\_conversion():

assert convert(0.001) == pytest.approx(149597870.691)

\*Floating-point arithmetic can sometimes lead to very small differences due to precision issues. `pytest.approx(149597870.691)` is used to allow a small margin of error in the comparison.

**def** test\_float\_conversion():

assert convert(0.001) == pytest.approx(149597870.691, abs=**0.1**)

\*`abs=0.1` This sets the tolerance level for the comparison. It means that the result of convert(0.001) should be within +/- `0.1` units of the expected value.

**def** test\_float\_conversion():

assert convert(0.001) == pytest.approx(149597870.691, abs=**1e-5)**

\*`abs=1e-5` This specifies that the difference between the actual result and the expected value should not exceed 1 x 10^-5 (or 0.00001).