**Python – unittest**

Create two files. One file containing the functions to be tested and the other file containing the test code. Both must be placed in same directory.

Create a class which inherits from unittest.TestCase, this will access to lot of testing capabilities.

In the test\_divide(self) function:

* self.assertRaises(ValueError, calc.divide, 10, 0) works perfectly because it is a value error.
* But if we pass self.assertRaises(ValueError, calc.divide, 10, 0) then it shows test has failed.
* To solve this we use a **context manager**:

with self.assertRaises(ValueError):

calc.divide(10, 0)

Consider a scenario where you have numerous test cases and in each case you have to create the same object. That same object can be created using two methods setUp() and tearDown().

setUp() runs its code before every single test, while tearDown() method runs its code after every single test.

Now you can place objects to be created before every test inside setup() method instead doing so for each test case. E.g:

def setUp(self):

print('setUp')

self.emp\_1 = Employee('Corey', 'Schafer', 50000)

self.emp\_2 = Employee('Sue', 'Smith', 60000)

In order to access these objects from other tests they must be set as instance attributes, which can be done using keyword self. And also everyehre we reference an object it must begin with self.

In the previous case methods were run prior to and after evry test case. You might also need something that runs once before all test cases and once after every test case is completed:

@classmethod

def setUpClass(cls):

print('setupClass')

@classmethod

def tearDownClass(cls):

print('teardownClass')

Consider a case where we need to test code by accessing some URL from the web. But for some reason if that website is down then the test code would throw an error. But we want it to throw an error only when there is an actual error in the code. So we use `mock` to get around this.

Tests must be independent of other tests.

By Socratica

unittest

For circle.py

from math import pi

def circle\_area(r):

if type(r) not in [int, float]:

raise TypeError("The radius must be non-negative real number")

if r < 0:

raise ValueError("The radius cannot be negative")

return pi\*(r\*\*2)

For test\_circle.py

import unittest

from circle import circle\_area

from math import pi

# following is a class of the sub-class TestCase class of

# unittest module

class TestCircleArea(unittest.TestCase):

def test\_area(self):

self.assertAlmostEqual(circle\_area(1), pi)

self.assertAlmostEqual(circle\_area(0), 0)

self.assertAlmostEqual(circle\_area(2.1), pi \* 2.1\*\*2)

def test\_area(self):

self.assertRaises(ValueError, circle\_area, -2)

def test\_types(self):

self.assertRaises(TypeError, circle\_area, 3+5j)

self.assertRaises(TypeError, circle\_area, True)

self.assertRaises(TypeError, circle\_area, "radius")

run the test modeule by python -m unittest test\_circles

-m -> tells python to run it as a script

One can also run python -m unittest

where python searches for test cases to be run using test discovery