# **Unit Testings**

def add(x, y):

return x + y

def subtract(x, y):

return x - y

def multiply(x, y):

return x \* y

def divide(x, y):

if y == 0:

raise ValueError('Can not divide by zero!')

return x / y

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import unittest

import calc

class TestCalc(unittest.TestCase):

def test\_add(self):

self.assertEqual(calc.add(2, 2), 4)

self.assertEqual(calc.add(-2, 2), 0)

self.assertEqual(calc.add(-2, -2), -4)

def test\_subtract(self):

self.assertEqual(calc.subtract(10, 5), 5)

self.assertEqual(calc.subtract(10, -5), 15)

self.assertEqual(calc.subtract(-10, 10), -20)

def test\_multiply(self):

self.assertEqual(calc.multiply(10, 5), 50)

self.assertEqual(calc.multiply(10, 0), 0)

self.assertEqual(calc.multiply(10, -1), -10)

def test\_divide(self):

self.assertEqual(calc.divide(20, 5), 4)

        # this's a method of testing exceptions

# self.assertRaises(ValueError, calc.divide, 10, 0) # first method

with self.assertRaises(ValueError): # my method of

calc.divide(10, 0)

if \_\_name\_\_ == "\_\_main\_\_": # means if file is running directly

unittest.main() # means run all of our test

Note 01:

Two methods of testing exceptions:

* self.assertRaises(ValueError, calc.divide, 10, 0)

This approach doesn’t call function normally. Namely, arguments of function are called separately as above.

But there is a better way of testing exceptions, see below:

* with self.assertRaises(ValueError):

calc.divide(10, 0)

This is a better approach to test exceptions. Function is called normally.

Note 02:

without code below:

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

unittest.main()

we need to run **unittest** as main module and run test\_calc.py file like below:

python –m unittest test\_calc.py # this means run unittest as main module

but since we are adding code below:

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

unittest.main()

we can simply run our python file as: python test\_calc.py

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

employee.py

class Employee:

    raise\_amount = 1.05

    def \_\_init\_\_(self, first, last, pay):

        self.first = first

        self.last = last

        self.pay = pay

    @property

    def fullname(self):

        return '{} {}'.format(self.first, self.last)

    @property

    def email(self):

        return self.first + '.' + self.last + '@gmail.com'

    def apply\_raise(self):

        self.pay = int(self.pay \* self.raise\_amount)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import unittest

from employee import Employee

class TestEmployee(unittest.TestCase):

    @classmethod

    def setUpClass(cls):   # setUpClass run its code BEFORE ALL TESTS

        print('Set up class (runs before all test methods)\n')

    @classmethod

    def tearDownClass(cls):  # tearDownClass run its code AFTER ALL TESTS

        print('Tear down class (runs after all test methods)\n')

    def setUp(self):  # setUp() method run its code before every single test

        print('setUp (run before every test)')

        self.emp\_1 = Employee('John', 'Doe', 2000)

        self.emp\_2 = Employee('Eric', 'Mathew', 3500)

    def tearDown(self):  # tearDown() method run its code after every test

        print('tearDown (run after every test) \n')

        pass

    def test\_email(self):

        print('test\_email')

        self.assertEqual(self.emp\_1.email, 'John.Doe@gmail.com')

        self.assertEqual(self.emp\_2.email, 'Eric.Mathew@gmail.com')

        self.emp\_1.first = 'Bradly'

        self.emp\_2.last = 'Cooper'

        self.assertEqual(self.emp\_1.email, 'Bradly.Doe@gmail.com')

        self.assertEqual(self.emp\_2.email, 'Eric.Cooper@gmail.com')

    def test\_fullname(self):

        print('test\_fullname')

        self.assertEqual(self.emp\_1.fullname, 'John Doe')

        self.assertEqual(self.emp\_2.fullname, 'Eric Mathew')

    def test\_applyRaise(self):

        print('test\_apply\_raise')

        self.emp\_1.apply\_raise()

        self.emp\_2.apply\_raise()

        self.assertEqual(self.emp\_1.pay, 2100)

        self.assertEqual(self.emp\_2.pay, 3675)

# After running above test

"""

Set up class (runs before all test methods)

.setUp (run before every test)

test\_apply\_raise

tearDown (run after every test)

setUp (run before every test)

test\_email

tearDown (run after every test)

.

setUp (run before every test)

test\_fullname

tearDown (run after every test)

.

Tear down class (runs after all test methods)

----------------------------------------------------------------------

Ran 3 tests in 0.001s

OK

 """

employee.py

import requests

class Employee:

raise\_amount = 1.05

def \_\_init\_\_(self, firstname, lastname, pay):

self.firstname = firstname

self.lastname = lastname

self.pay = pay

# print("Employee created: {} - {}".format(self.fullname, self.email))

@property

def fullname(self):

return "{0} {1}".format(self.firstname, self.lastname)

@property

def email(self):

return "{}{}@gmail.com".format(self.firstname, self.lastname)

def apply\_raise(self):

self.pay = int(self.pay\*self.raise\_amount)

def monthly\_schedule(self, month):

# testing monthly\_schedule in below page

request = requests.get(f'http://company.com/{self.lastname}/{month}')

print(request) # <Response [200]>

if request.ok:

return request.text

else:

return "bad request"

emp\_1 = Employee('John', 'Doe', 100000)

emp\_1.monthly\_schedule('June')

**[Mocking is not a must learn subject]**

import unittest

from employee import Employee

from unittest.mock import patch

class TestEmployee(unittest.TestCase):

@classmethod

def setUpClass(cls): # def ! class

print("setUpClass")

@classmethod

def tearDownClass(cls):

print("tearDownClass")

def setUp(self):

print("setUp")

self.emp\_1 = Employee('John', 'Doe', 5000)

self.emp\_2 = Employee('Mike', 'Smith', 6000)

def tearDown(self):

print("tearDown\n")

pass

def test\_emp\_fullname(self):

print("fullname")

self.assertEqual(self.emp\_1.fullname, 'John Doe')

self.assertEqual(self.emp\_2.fullname, 'Mike Smith')

self.emp\_1.firstname = 'Jane'

self.emp\_2.firstname = 'Logan'

self.assertEqual(self.emp\_1.fullname, 'Jane Doe')

self.assertEqual(self.emp\_2.fullname, 'Logan Smith')

def test\_emp\_email(self):

print("email")

self.assertEqual(self.emp\_1.email, 'JohnDoe@gmail.com')

self.assertEqual(self.emp\_2.email, 'MikeSmith@gmail.com')

self.emp\_1.firstname = 'Jane'

self.emp\_2.firstname = 'Logan'

self.assertEqual(self.emp\_1.email, 'JaneDoe@gmail.com')

self.assertEqual(self.emp\_2.email, 'LoganSmith@gmail.com')

def test\_raise\_amount(self):

print("raise\_amount")

self.emp\_1.apply\_raise()

self.emp\_2.apply\_raise()

self.assertEqual(self.emp\_1.pay, 5250)

self.assertEqual(self.emp\_2.pay, 6300)

def test\_employee\_monthly\_schedule(self):

# patch 'requests.get' of 'employee' module

# we are mocking request.get

with patch('employee.requests.get') as mocked\_get:

mocked\_get.return\_value.ok = True

mocked\_get.return\_value.text = "Success"

schedule = self.emp\_1.monthly\_schedule('Nov')

mocked\_get.assert\_called\_with('http://company.com/Doe/Nov')

self.assertEqual(schedule, 'Success')

mocked\_get.return\_value.ok = False

schedule = self.emp\_2.monthly\_schedule('June')

mocked\_get.assert\_called\_with('http://company.com/Smith/June')

self.assertEqual(schedule, 'bad request')

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

unittest.main()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**What Is Mocking?**

Mocking is a library for testing in Python which allows you to replace parts of your system under test with mock objects and make assertions about how they have been used. In Python, mocking is accomplished by replacing parts of your system with mock objects using the **unittest**.

**assert\_called\_with()**

'This method is a convenient way of asserting that calls are made in a particular way'

so it tests whether the parameters are being used in the correct way.