# Python : Test

Learn to Code with Python

# Unit Testing – Assert Statement

- assert → If the assertion is not met ( evaluates to true ) then it throws AssertionError else evalautes to None
  - Parameters
    - 1st argument evaluates to false
    - 2nd A message print if AssertionError happens
  - Assert is a keyword so you not need parenthesis
- example
  - def add(x,y):
    - assert isinstance(x,int) and isinstance(y,int), Both Argument should be > 0”
    - return x + y

# Unit Testing DocTest Module

- Search the doc strings for python expression
  - A function is called and confirms the python expression found the doc string has the correct return value as found in the doc string
  - unreliable – super literal example : “chuck” is considered different than ‘chuck’
- Example
  - def sum_of_list(numbers)
    - “”” Return the sum of all numbers in a list
    - >>> sum_of_list([1,2,3])
    - 6
    - >>> sum_of_list([1,2,3])
    - 26
    - “””
    - total = 0
    - for num in numbers:
      - total += num
    - return total
    - if __name__ == ‘__main__’
      - import doctest
      - doctest.testmod()  // Finds python express and evaluates them.  If there is not a match get a Test Failed message

# Unit Testing UnitTest Module assertEqual Method

- Example
  - import unittest
  - assertEqual → Accepts two arguments and validates they are the same
  - When the python script is executed For each passing test executed a dot is printed out then Ran < test count> in <number> secs
    - When a test fails a dot is not displayed but an F
    - If using assertEqual then you will also get an AssertionError the application will execute the next test
  - class TestStringMethods(unittest.TestCase):
    - def test_split(self): // The method must have the name test as part of the name so the test will be called
      - self.assertEqual(“a-b-c”.split(“-”), [“a”, “b”, “c”])
    - def test_count(self):
      - self.assertEqual(“beautiful”.count(“u”), 2)
  - if __name__ = = ‘__main__’
    - unittest.main()
  - Convention: Write one assertion per test

# Unit Testing: Skipping Test and Expected Failures

- Example
  - import unittest
  - class TestSkippingStuff(unittest.TestCase)
    - def test_addition(self):
      - self.assertEqual(1 + 1, 2)
    - def test_subtraction(self):
      - self.assertEqual(10 - 5,5)
    - @unittest.skip (“To Be implemented later”) // The unittest will not be executed
    - def test_multiplication(self):
      - pass // Even if skipped the body must have code
  - if __name__ == “__main__”:
    - unittest.main()
- The output of unittest.main() will bge .s. where the s means the test has been skip

# Unit Testing: Assertions – assertNotEqual and Custom Error Message

- The Custom Error will always be the last argument. It could be the 3rd, 10th
- Example
  - import unittest
  - def copy_and_add_element(values,element):
    - copy = values[:]
    - copy.append(element)
    - return copy
  - class TestInequality(unittest.TestCase):
    - def test_inequality(self):
      - self.assertNotEqual(1,2)
      - self.assertNotEqual(True, False)
      - self.assertNotEqual("Hello", "hello")
      - self.assertNotEqual([1,2], [2,1])
    - def test_copy_and_add_an_element(self):
      - values = [1,2,3];
      - result = copy_and_add_an_element(values,4):
      - self.assertEqual(result, [1,2,3,4])
      - self.assertNotEqual( values, [1,2,3,4], " The output list is the same as the input") // Example of a custom message
  - if __name__ == "__main__":
    - unittest.main()

# Unit Testing – Test Object Identity with Asserts amd assertsNot

- Example
  - import unittest
  - def IdentityTests(unitTest.TestCase):
    - def test_identically(self):
      - a [1,2,3]
      - b = a
      - c = [1,2,3]
      -
      - self.assertEqual(a,b) // True point to the same object
      - self.assertEqual(a,c) // True use equal which compares the values in the object
      - self.assertIs(a,b) // Are they same object → True
      - self.assertIs(a,c) // Are they same object → False
      -
  - if __name__ == “__main__”:
    - unittest.main()

# Unit Testing – Test Truthiness and Falsiness

- Example
  - import unittest
    - def IdentityTests(unitTest.TestCase):
      - def test_truthiness(self):
        - self.assertTrue(3 < 5) // Does not evaluate Truth or False, but Truthiness.
        - self.assertTrue(1)
        - self.assertTrue(“hello”)
        - self.assertTrue([“a”])
        - self.assertTrue(“{ “b” : 5 })
      - def test_falsiness(self):
        - self.assertFalse(False)
        - self.assertFalse(0)
        - self.assertFalse(“”)
        - self.assertFalse([])
        - self.assertFalse({})
- if __name__ == “__main__”:
  - unittest.main()

# Unit Test – Assertions Test Nullness (AssertNone and AssertNotNone)

- Example
  - import unittest
  - def explicit_return_sum(a,b):
    - return a + b
  - def implicit_return_sum(a,b):  // The function has not return so it returns
    - print a + b
  - class TestNone(unittest.TestCase):
    - def test_sum_functions()
      - self.assertIsNone( implicit_return_sum(3,5))  // Will return True
      - self.assertIsNotNone(explicit_return_sum(3,5))  // Will return True
  - if __name__ == “__main__”):
    - unittest.main()
- Don’t forget everything in Python is an object even None

# Unit Test – Assertions Test inclusion with assertin assertNotIn

- Example
  - import unittest
  - class InclusionTests(unittest.TestCase):
    - def test_inclusion(self):
      - self.assertIn(“k”, “king”)
      - self.assertIn(1, [1,2,3])
      - self.assertIn( “a”, { “a”:1 “b”:2 } )
      - self.assertIn(55, range(50,59))
    - def test_noninclusion(self):
      - self.assertInNotIn(“w”, “king”)
      - self.assertInNotIn(10, [1,2,3])
  - if __name__ == “__main__”):
    - unittest.main()

# Unit Test – Test Object Type ( assertinstance and assertNotInstance)

- Example
  - import unittest
  - class ObjectTypeTest(unittest.TestCase):
    - def test_inclusion(self):
      - self.assertIsInstance(1, int)
      - self.assertIsInstaince( { “a”:1}, dict)
    - def test_noninclusion(self):
      - self.assertNotIsInstance(5, dict)
      - self.assertNotIsInstance({“a”:1}, list)
      -
  - if __name__ == “__main__”):
    - unittest.main()

# Unit Test – The Setup and Tear Down Methods

- Fixture : A piece of code that constructs and configures an object or system under test
- Example
- import unittest
- class Address():
  - def __init__(self, city, state):
    - self.city = city
    - self.state = state
- class Owner():
  - def __init__(self, name, age):
    - self.name = name
    - self.age = age
- class Restaurant():
  - def __init__(self,address,owner):
    - self.address = address
    - self.owner = owner
  - @property
  - def owner_age(self):
    - return self.owner.age
  - def summary(self):
    - return f”This restaurant is ownd by {self.owner} and is located in { self.address}

# Unit Test – The Setup and Tear Down Methods

- The developer should execute the test to be called in order
- setUp and TearDown execute before and after each test(s)
  - use self so the other test functions can access the data created in the setup
- Example continued from last page
  - class TestRestaurant(unittest.TestCase):
    - def setUp(self):
      - address = Address(city = “New York”, state = “New York”)
      - owner = Owner( name = “Jackie”, age = 60)
      - self.restaurant = Restaurant( address, owner) // Instantiating a new Restaurant Object each time
    - def tearDown(self):
      - pass
    - def test_owner_age():
      - self.assertEqual( self.golden_palace.owner_age, 60)
  - if __name__ == “__main__”):
    - unittest.main()

# The SetupClass and tearDownClass Methods

- setUpClass will run when the test suite is started and tearDownClass will run when the Test Suite is completed
  - Meant to be used for costly operations such as connecting to a database and then destroying the connection
  - Have to use the name tearDown and tearDownClass, setupClass and setup those functions are overriding the functions in the parent class
- Example
  - import unittest
    - class TestOperations(unittest.TestCase):
      - @classMethod
      - def setupClass(cls):
        - print(“This will run once before the test suite starts”)
      - @classMethod
      - def tearDownClass(cls)
        - print(“This will run once after the test suite ends”)
      - def setup(self):
        - print(“This will run before each test”)
      - def tearDown(self):
        - print(“This will run after EACH test”)
      - def test_stuff(self):
        - self.assertEqual(1,1)
      - def test_more_stuff(self):
        - self.assertEqual([],[])
- 
  - if __name__ == “__main__”):
    - unittest.main()

# Unit Test : Assertions Errors with assert Raises

- Test to verify a chunk of code raises an error.  If an exception is raised in the test routines  the TestRunner could crash
- assertRaises
  - parameter 1 — Is the exception that is expect to be thrown
  - parameter 2 — Is the function that will be executed
  - parameter 3 to number of arguments - 1 — parameters to the function found in “parameter 2”
  - parameter 4 — optional custom message
- with – Allows the TestRunner to continue to run even if a specific problem is raised
- Example
  - import unittest
  - def divide(x,y):
    - if y == 0:
      - raise ZeroDivisionError
    - return x / y
  - class ObjectTypeTest(unittest.TestCase):
    - def test_divide(self):
      - self.assertRaises( ZeroDivisionError, divide, 3, 0
    - def test_divide_another_way(self):
      - with self.assertRaises(ZeroDivisionError):
        - divide(10,0)
- 
  - if __name__ == “__main__”):
    - unittest.main()

# Mocking : Intro to Mocking with Mock Class

- A mock is an object that takes the place of another object in a test
- Use mock object to allow the unit test to only test one piece of our system
  - The mock serves as a substitute
  - Allows use to verify that an error more likely coming from the class we are testing
- Example
  - from unittest.mock import Mock
  - pizza = Mock() // Assign attributes dynamically add attributes
  - print(pizza) // < Mock id='449833345984'>
  - print(type(pizza)) // unittest.mock.Mock
  - pizza.size = "Large"
  - pizza.price = "19.99"
  - pizaa.toppings = [ "Pepperoni", "Mushroom", "Saurage" ]
  - print(pizza.size)
  - print(pizza.price)

# Mocking : Intro to Mocking with Mock Class

- Example – from previous
  - anchovies does not exit on pizza, but you will get back an object
  - print(pizza.anchovies) // Returns <Mock name = ‘mock.nonsense’ id=’4508054184’ >
  - print(pizza.anything.we.want) // Can mock nest structures
  - pizza.cover_with_cheese() // returns <Mock name=’mock.cover_with_cheese()’ id=’4514286032’>
- pizza.configure_mock( // A actual function that initialize a mock
  - size = “Large”,
  - price = 19.99
  - toppings = [ “Pepperoni”, “Mushroom”, “Sausage” ]
- )

# Mocking: The return_value attribute

- A mock object can be called and invoked with a pair of parenthesis
- Example
  - from unittest.mock import Mock
  -
  - mock = Mock()
  - print(mock()) // returns < Mock name-’mock()’ id=’4481243312’>
  -
  - mock.return_value = 25
  - print(mock()) // prints 25
  -
  - mock = Mock(return value = 25)
  - print(mock()) // prints 25
  -
  -

# Mocking: The return_value attribute

- Example
  - stuntman = Mock()
  - stuntman.jump_of_building.return_value = “Oh No my leg”
  - stuntman.light_on_fire.return_value = “My Leg”
  - print(stuntman.jumpOf_building() ) // prints “Oh No my Leg

# Mocking: The Side Effect Attribute

- Example
  - Use when we return value to be more dynamic
  - from unittest.mock import Mock
  - form random import randint
  -
  - def generate_number():
    - return randint(1,10)
  - call_me_maybe = Mock()
  - pirnt(call_me_maybe.side_effect) // Returns none, can be assigned an iterable, a function, an exception
  - call_me_maybe.side_effect = generate_number
  - print(call_me_maybe()) // The function generate_number() will be invoked
  - call_me_maybe = Mock(side_effect = generate_number) // another way of assigning the side effect
  - call_me_maybe = Mock(return_value = 10 side_effect = generate_number) // Only the side_effect value will be returned here
  - Use : Your mock is going to be called several times and you want each to a different return value

# Mocking: The Side Effect Attribute

- Example – Emulate a pop method on a list  call pop and return a value.  When pop is called on an empty list throw IndexError
  - three_item_list = Mock()
  - three_item_list.pop.side_effect = [3,2,1, IndexError(“pop from empty list”)]
  - print(three_item_list.pop())  // returns 3
  - print(three_item_list.pop())  // returns 2
  - print(three_item_list.pop())  // returns 1
  - print(three_item_list.pop())  // throws IndexError
- We can have the same functionality with a list without dealing with the implementation
- mock = Mock(side_effect = NameError(“Some error message”)
- mock()  // Raise an exception
- mock.side_effect = None  // resets the side affect and a mock object is returned when called

# Mocking: The Mock Object vs MagicMock Objects

- A subclass of Mock with default implementations of most of the magic methods
- MockObject does not support magic methods ( __len__ , __repr__ ).  Should use MagicMock by default
- MagicMock supports indexing while Mock does not
- Example
  - plain_mock = Mock()
  - magic_mock = MagicMock()
  - print(len(plain_mock)) // TypeError : object of type ‘Mock’ has no len
  - print(len(magic_mock)) // return 0
  - print(main_mock[3]) // Get MagicMock name = mock.__get__item()’ id=’4497187120’ ( Returns same object for all // indexes
  - magic_mock.__len__ // Returns a Mock.  Every attribute on a mock is going to be another mock
  - magic_mock.__len__.return_value = 50
  - print(len(magic_moc)) // Returns 50

# Mocking: The Mock Object vs MagicMock Objects

- A subclass of Mock with default implementations of most of the magic methods
- MockObject does not support magic methods ( __len__ , __repr__ ).  Should use MagicMock by default
- MagicMock supports indexing while Mock does not
- Example
  - plain_mock = Mock()
  - magic_mock = MagicMock()
  - print(len(plain_mock)) // TypeError : object of type ‘Mock’ has no len
  - print(len(magic_mock)) // return 0
  - print(main_mock[3]) // Get MagicMock name = mock.__get__item()’ id=’4497187120’ ( Returns same object for all // indexes
  - magic_mock.__len__ // Returns a Mock.  Every attribute on a mock is going to be another mock
  - magic_mock.__len__.return_value = 50
  - print(len(magic_moc)) // Returns 50

# Mocking:Mock Calls

- How we can ask a mock has been, how its been invoke and how many times
- Examle
  - import unnitest
  - from unittest.mock import MagicMock
  - class MockCallsTest(unittest.TestCase):
    - def test_mocks_calls(self):
      - mock = MagicMock()
      - mock()
      - mock.assert_called()  // An actual method which asserts that the mock was call or invoked at least once
    - def test_mocks_calls(self):
      - mock = MagicMock()
      - mock.assert_called()  // Here it fails
    - def test_not_called(self):
      - mock = MagicMock()
      - mock.assert_not_called()
    - def test_called_with(self):
      - mock = Maigic
      - mock(1,2,3)
      - mock.assert_called_with(1,2)  // Failure called with 1 and 2 and 3 , but assert looking for 1 and 2 only

# Mocking:Mock Calls

- Example Continued
  - def test_mock_attributes(self):
    - mock = MagicMock()
    - mock()
    - mock(1,2)
    - print(mock.called) // True
    - print(mock.call_count) // 2 since mock() and mock()
    - print(mock.call_count) // [ call(), call(1,2) ]

# Mocking : Putting it all together

- If we have object dependent on each other and the test breaks we can never be sure which object is responsible.
  - Mocks allow use to create separation of objects for test
- import unittest
- from unittest.mock import MagicMock
- class Actor():
  - def jump_out_of_helicopter(self):
    - return “Nope, not doing it!”
  - def light_on_fire(self):
    - return “Heck no, where’s my agent?”
- class Movie():  // Difficult to test Movie in isolation since it depends on actor
  - def __init__(self, actor):
    - self.actor = actor
  - def start_filming(self):  // Actor can be a mock object.  Need to verify that both methods have been invoked
    // Verify that the methods have been called.  Not worried about the internals of the function called.
    // The values of the functions called is the responsibility of another test
    - self.action.jump_out_of_helicopter()
    - self.action.light_on_fire()

# Mocking : Putting it all together

- class MovieTest(unittest.TestCase):
  - def test_start_filming(self):
    - stuntMan = MagicMock()
    - movie = Movie(stuntMan)
    -
    - movie.startFilming()
    - stuntman.jump_out_of_helicopter.assert_called()
    - stuntman.light_on_fire.assert_called()
- if __name__ == “__main__”:
- unittest.main()

# Mocking: Verifying Doubles

- Ultimate Object of an mock is to play the role of the real class
  - Problem : There is no guarantee they match the entities they are suppose to mock.
- Example
  - from unittest.mock import MagicMock
  - class BurrioBowl():
    - restaurant_name = “Bobo’s Burritos
    - @clasmethod
    - def steak_special(cls);
      - return cls(“Steak”, “White”, 1)
    - def __init__(self, protein, rice, guacamole_portions):
      - self.protein = protein
      - self.rice = rice
      - self.guacamole = guacamole_portions
    - deff add guac(self):
      - self.guacamole_portions += 1

# Mocking: Verifying Doubles

- MagicMock spec parameter
  - Cause the mock to have the information about the BurritoBowl class
  - If we attempt to access method/properties on the class that are not on the Actual Class we have an exception
  - The spec is not strict you can add mock properties/functions to the mock
- Example Continued
  - class_mock = MagicMock(spec = BurritoBowl) // spec that will resemble a burio bowl class.
  - print(class_mock.steak_special) // Returns an MockObject ( Success )
  - print(class_mock.chicken_special) // Get an AttributeError. Can only access Class Variables/Funciton
    // Can only access restaurant name or steak_special
  - instance_mock = MagicMock( spec = BurritoBowl.steak_special() ) // Creates an BurritoBowl Object
  - print(instance_mock.protein) // Success
  - print(instance_mock.add_guac()) // Will call the function, but not access it.
  - print(instance_mock.add_cheese()) // Throws attributeError addCheese
  - instance_mock.beans = True // Add to the mock class
  - print(instance_mock) // Returns true. The spec is not strick you can add mock properties/functions
  - instance_mock = MacgicMock(spec_set = BurritoBowl.steak_special()) // Follows the instance object very strictly cannot add instances/methods

# Mocking: Patch 1 : The Patch Function

- Can be used as a function or decorator that will automatically create a magic mock object in place of an existing object in some module
- Works by temporarily changing the object that the name points to with another one
  - intercept a call to a real object and automatically put a magic mock object in that place instead
- Use Case : Mock out deep nested hierarchies that belong in other modules outside our code.
- Example
  - import urllib.request
  - class webRequest():
    - def __init__(self, url):
      - self.url = url
    - def execute(self):
      - response = urllib.request.urlopen(self.url)
      - “SUCCESS” if response.status == 200 else “FAILURE”
  - wr = WebRequest(“www.google.com”)
  - wr.execute()
- Three problems with testing this small unit individually
  - Depends on urlopen() works and how it was implemented
  - Depends on the internet being up and  google site being accessible

# Mocking: Patch 1 : The Patch Function

- The with keyword create a block where some data will exist for the lifetime of the block
- A unit test is suppose to be small light weight and independent
  - For unit test never reach out to a website, access a database or reach to your computer’s file system they are slow and flakey.
- When you create separation between a unit and the external dependencies the test are usually faster and more stable
  - Each dependency is something you are reliant one.
- Goal : To use a mock object to fake the response object that is returned.
  - Inside the body of the function or with statement the target is patched with a new object.
- Example continued from previous Example
  - import unittest
  - from unittest.mock import patch

# Mocking: Patch 1 : The Patch Function

- class WebRequestTest(unittest.TestCase):
  - def test_execute_with_success_repsonse(self):
    - with patch('urllib.request.urlopen') as mock_urlopen // The variables (mock_urlopen) is a magic mock
      - mock_urlopen.return_value.status = 200
      - wr = WebRequest("http://www.google.com")
      - assertEqual( wr.execute(), "SUCCESS") // The urlopen is not begin called, but the mock is return the value
  - def test_execute_with_failure_response(self):
    - with patch('urllib.request.urlopen') as mock_urlopen // The variables (mock_urlopen) is a magic mock
      - mock_urlopen.return_value.status = 404
      - wr = WebRequest("http://www.google.com")
      - assertEqual( wr.execute(), "FAILURE") // The urlopen is not begin called, but the mock is return the value
- 
- if __name__ == "__main__"
  - unittest.main()

# Mocking: Patch 2: The @Patch Operator

- Can also use patch as a decorator
- Example – See previous example
- @patch('urllib.request.urlopen') // Creates a magic mock for urllib.request.urlopen and to the def add mock_urlopen
- class WebRequestTest(unittest.TestCase):
  - def test_execute_with_success_repsonse(self, mock_urlopen):
    - with patch('urllib.request.urlopen') as mock_urlopen // The variables (mock_urlopen) is a magic mock
    - mock_urlopen.return_value.status = 200
    - wr = WebRequest("http://www.google.com")
  - assertEqual( wr.execute(), "SUCCESS") // The urlopen is not begin called, but the mock is return the value

# Mocking: Patch 3 : What Patch Patches

- The patch function being mocked should be looked up and not defined.
- Example
  - urlopen is defined urllib.request.urlopen, but you want directly use it from your name space self.urlopen
  - from urllib.request import urlopen which puts it into your current namespace
    - The way the test is written the test will fail since the magic mock is being created from urllib.request.openurl
    - need to change it webrequest.urlopen ( self.webrequest.urlopen ) is the