Implementation and Unittests of AWS, Google Storage (Cloud) and Amazon S3 Functions using Mock, Boto and MagicMock in Python

Swati Sinha[1], Preetam Pati[2]

[1]*Application Development Analyst in Accenture Services Pvt. Ltd. Gurugram, Haryana, India*
[2]*Student, GreatLakes Institute of Management, Gurugram, Haryana, India*

---***---

Abstract - *Earlier, we ran such service tests against the service to be tested, which connects against managed backend services as such AWS S3. Doing so, we faced some disadvantages. Firstly, these tests are potentially long-running as they depend directly on managed services that include network latency, especially when running the build and tests on a local developer machine. Secondly, tests rely on the availability and reachability of managed services. If a test run, for whatever reason, cannot access managed services the build will fail as a result and we're potentially not able to roll out new versions. A mock object is meant to quickly and easily represent some complex object without having to manually go through and set up stubs for that object during a test, which overcome above problems. Anything that is external to the module-under-test should be mocked. Amazon S3 is an object storage service which can be created, configured and managed with Boto3, which is an AWS SDK for Python. In other words, it is used to interact with AWS. It provides an easy to use, object-oriented API as well as low-level access to AWS services (e.g. class S3.Client- A low-level client representing Amazon S3). There are various methods available inside this class which will be used directly to handle the migration of data. And hence these methods can be tested using mock (mock_s3) i.e. we are mocking S3 services. We are using the decorator @mock_s3 to specify we want to mock out all the calls to S3.*

***Key Words*: mock, patch, unittest, side_effect, MagicMock, s3, moto, boto3**

1.INTRODUCTION

unittest.mock is a library for testing in Python, not a framework. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used. unittest.mock provides a core Mock class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods/attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way. Additionally, mock provides a patch() decorator that handles patching module and class level attributes within the scope of a test, along with sentinel for creating unique objects Mock is very easy to use and is designed for use with unittest. Mock is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks. In other words, it works in AAA (Arrange Act Assert) rather than record and replay. There is a backport of unittest.mock for earlier versions of Python, available as mock as PYPI.

Mock and MagicMock objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```
>>> from unittest.mock import MagicMock

>>> thing = ProductionClass()

>>> thing.method = MagicMock(return_value=3)

>>> thing.method(3, 4, 5, key='value')

3

>>> thing.method.assert_called_with(3, 4, key='value')side
_effect allows you to perform side effects, including raising an
exception when a mock is called:
>>> mock = Mock(side_effect=KeyError('foo'))

>>> mock()

Traceback (most recent call last):

 ...

KeyError: 'foo'

>>> values = {'a': 1, 'b': 2, 'c': 3}
                         >>> def side_effect(arg):

...     return values[arg]

...

>>> mock.side_effect = side_effect

>>> mock('a'), mock('b'), mock('c')

(1, 2, 3)
```

```
>>> mock.side_effect = [5, 4, 3, 2, 1]

>>> mock(), mock(), mock()

(5, 4, 3)
```

Why was MagicMock made a separate thing rather than just folding the ability into the default mock object?

Ans - One reasonable answer is that the way MagicMock works is that it preconfigures all these protocol methods by creating new Mocks and setting them, so if every new mock created a bunch of new mocks and set those as protocol methods and then all of those protocol methods created a bunch more mocks and set them on their protocol methods, you've got infinite recursion...

What if you want accessing your mock as a container object to be an error -- you don't want that to work? If every mock has automatically got every protocol method, then it becomes much more difficult to do that. And also, MagicMock does some of this preconfiguring for you, setting return values that might not be appropriate, so I thought it would be better to have this convenience one that has everything preconfigured and available for you, but you can also take an ordinary mock object and just configure the magic methods you want to exist...

The simple answer is: just use MagicMock everywhere if that's the behavior you want. One important point to add MagicMock is iterative.

1.1 Advantages of Mocking

- **Avoiding Too Many Dependencies.** Mocking reduces the dependence of functions. For instance, if you have a function A class that depends on a function B, you will need to write a few unit tests covering the features provided by function B. Let's say the code grows in future and you have more functions, i.e. A depends on B, B depends on C, and C depends on D. If a fault is introduced in Z, all your unit tests will fail.
- **Reduced overload.** This applies to resource-intensive functions. A mock of that function would cut down on unnecessary resource usage during testing, therefore reducing test run time.
- **Bypass time constraints in functions.** This applies to scheduled activities. Imagine a process that has been scheduled to execute every hour. In such a situation, mocking the time source lets you actually unit test such logic so that your test doesn't have to run for hours, waiting for the time to pass.

1.2 When to Mock

1.21 Only mock types that you own

External types have dependencies on their own. I also might not fully understand how they work, and they might even change their behavior in a next version. Mocking third-party code can be a problem

1.22 Don't mock values

If an object is mocked to return it from the service. This type is a value. It has no identity and its fields are most probably immutable. Values should not be mocked. Because, mocking is a technique that is used to make the relationships and interactions between objects visible. It's not a tool to make it easier to instantiate complex objects. Getting the value of a price is not an interaction.

1.23 Avoid mocking concrete classes

If an object has five public methods, but current object under test is using only two of them (the Interface Segregation Principle is violated), it's a good indicator that we need to create another type. This extra relationship is harder to spot when we use concrete objects instead of interfaces. Additionally, if we mock a method of an object, but forget a different method that the object under test also calls, the test might fail with obscure errors.

1.3 Application of Mocking

1.3.1 Various ways of introspect

```
>>>mock = Mock(name='bar', return_value='fish')

>>>mock(1,7, spam=99)

'fish'

>>>mcok.assert_called_once_with(1,7, spam =99)

>>>mock.called

True

>>mock.call_count

1

>>>mock.call_args

((1,7), {'span': 'eggs'})

>>>mock = Mock(return_value=None)

>>>mock(1,2,3)

>>>mock.assert_called_with(1, 2, 3)

>>>mock(4, 5, 6)

>>>mock.call_Args_list
```

```
[((1, 2, 3), {}), ((4, 5, 6), {})]

>>> mock.reset_mock()

>>>mock.call_args_list

[]
```

.

side_effect is passed to a function, it is going to be called in same argument as the mock is called with. Dynamic stuff can be done using side-effects. If side effect is an instance of an exception or an exception is called, mock is called exception is raised

1.32 MagicMock is even better

MagicMock is a class. There is python protocol method which require return value of specific type or cast return value to specific type, so MagicMock has all these features configured. You can access dictionary, return mock object, power, divided by mock.

```
>>>mock = MagicMock()

>>> mock['foo']

<mock.Mock object at 0x…>

>>> mock ** mock / mock

<mock.Mock object at 0x…>

>>> int(mock), float(mock), hex(mock)

(1, 1.0, '0x1')

>>> with mock:

---- pass

…

>>> list(mock)

[ ]
```

1.33 Magic Method Assertions

```
>>> mock = MagicMock()

>>> mock['foo'] = 3

>>> mock.__setitem__.assert_called_with('foo', 3)
```

```
>>> int(mock)

1

>>> mock.__int__.call_count
1
```

2 USAGE OF MOTO

Code to test:

```
import boto
from boto.s3.key import Key

class MyModel(object):
  def __init__(self, name, value):
    self.name = name
    self.value = value

  def save(self):
    conn = boto.connect_s3()
    bucket = conn.get_bucket('mybucket')
    k = Key(bucket)
    k.key = self.name
    k.set_contents_from_string(self.value)
```

2.1 Decorator

With a decorator wrapping, all the calls to S3 are automatically mocked out.

```
import boto
from moto import mock_s3
from mymodule import MyModel
@mock_s3
def test_my_model_save():
  conn = boto.connect_s3()
  # We need to create the bucket since this is all in Moto's
'virtual' AWS account
  conn.create_bucket('mybucket')
  model_instance = MyModel('steve', 'is awesome')
  model_instance.save()
  assert
conn.get_bucket('mybucket').get_key('steve').get_contents
_as_string() ==
  ↪'is awesome'
```

2.2 Context manager

Same as the Decorator, every call inside the with statement is mocked out.

```
def test_my_model_save():
  with mock_s3():
    conn = boto.connect_s3()
    conn.create_bucket('mybucket')
    model_instance = MyModel('steve', 'is awesome')
    model_instance.save()
    assert
conn.get_bucket('mybucket').get_key('steve').get_contents
_as_string()
    ‹→== 'is awesome'
```

2.3 Raw

You can also start and stop the mocking manually

```
def test_my_model_save():
  mock = mock_s3()
  mock.start()
  conn = boto.connect_s3()
  conn.create_bucket('mybucket')
  model_instance = MyModel('steve', 'is awesome')
  model_instance.save()
  assert
conn.get_bucket('mybucket').get_key('steve').get_contents
_as_string() ==
  ‹→'is awesome'
  mock.stop()
```

2.4 Stand-alone server mode

Moto also comes with a stand-alone server allowing you to mock out an AWS HTTP endpoint. For testing purposes, it's extremely useful even if we don't use Python. $ moto_server ec2 -p3000 * Running on http://127.0.0.1:3000/ However, this method isn't encouraged if we are using boto, the best solution would be to use a decorator method

3 S3 FUNCTIONS AND THEIR UNIT TESTS

3.1 Introduction to S3 and code for various function

Amazon Simple Storage Service is storage for the Internet. It is designed to make web-scale computing easier for developers. Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

```
def upload_file(file, bucket, key, raise_exception=False,
nolog=False,

        extra_args=None):

  uri = "s3://%s/%s" % (bucket, key)

  if not isinstance(extra_args, dict):

    extra_args = {}

  try:

    s3 = boto3.client('s3')

    s3.upload_file(file, bucket, key, ExtraArgs=extra_args)

  except Exception as e:

    if raise_exception is True:

      raise

    if not nolog:

      logging.error('unable to upload %s to %s: %s' %
(file, uri, e))

    return False

  if not nolog:

    logging.info('uploaded %s' % uri)

  return True



def upload_object(bucket, key, obj_str,
raise_exception=False, nolog=False,

        extra_args=None, encrypt_key=None):

  if not isinstance(extra_args, dict):

    extra_args = {}

  if key.endswith('.gz'):

    obj_str = gzip_string(obj_str)

  if encrypt_key is not None:

    obj_str = encrypt_decrypt('encrypt', encrypt_key,
obj_str)

  kwargs = {

    'Body': obj_str

  }

  kwargs.update(extra_args)
```

```
    uri = "s3://%s/%s" % (bucket, key)
    try:
        s3 = boto3.resource('s3')
        s3.Object(bucket, key).put(**kwargs)
    except Exception as e:
        if raise_exception is True:
            raise
        if not nolog:
            logging.error('unable to upload object to %s: %s' %
(uri, e))
        return False
    if not nolog:
        logging.info('uploaded %s' % uri)
    return True


def list_objects(bucket, prefix, delimiter='/', limit=None,
transform=None,
                 strip_prefix=False, return_prefixes=False):
    client = boto3.client('s3')
    kwargs = {
        'Bucket': bucket,
        'Delimiter': delimiter,
        'Prefix': prefix
    }
    if isinstance(limit, int):
        kwargs['MaxKeys'] = limit
    keys = []
    prefix_len = len(prefix)
    prefixes = {}
    while True:
        response = client.list_objects(**kwargs)
        marker = response.get('NextMarker')
        for d in response.get('Contents', []):
            if strip_prefix is True:
                d['Key'] = d['Key'][prefix_len:]
            if transform is not None:
                keys.append(jmespath.search(transform, d))
            else:
                keys.append(d['Key'])
        for d in response.get('CommonPrefixes', []):
            if strip_prefix is True:
                d['Prefix'] = d['Prefix'][prefix_len:]
            if d['Prefix'] != '/':
                prefixes[d['Prefix']] = True
        if marker is not None:
            kwargs['Marker'] = marker
        else:
            break
    if return_prefixes is True:
        return (keys, sorted(prefixes.keys()))
    return keys



def object_exists(bucket, key, modified_since=None,
return_response=False,
                  raise_exception=False):
```

```
    client = boto3.client('s3')
    response = None
    uri = 's3://%s/%s' % (bucket, key)
    kwargs = {
        'Bucket': bucket,
        'Key': key
    }

    if isinstance(modified_since, str):
        modified_since = parse_date(modified_since)

    if isinstance(modified_since, datetime):
        kwargs['IfModifiedSince'] = modified_since
    try:
        response = client.head_object(**kwargs)
    except botocore.exceptions.ClientError as e:
        code = e.response['Error']['Code']
        if code in ['NoSuchKey', '404']:
            if raise_exception is True:
                raise Exception('S3 URI %s not found' % uri)
            return False
        elif code in ['NotModified', '304']:
            return False
        else:
            raise Exception('unable to head_object %s: %s' % (uri, e))
    if return_response is True:
        return response
    else:
        return True


def download_file(bucket, key, file, raise_exception=False, nolog=False):
    uri = "s3://%s/%s" % (bucket, key)
    try:
        s3 = boto3.client('s3')
        s3.download_file(bucket, key, file)
    except Exception as e:
        if raise_exception is True:
            raise
        if not nolog:
            logging.error('unable to download %s to %s: %s' % (uri, file, e))
        return False
    if not nolog:
        logging.info('downloaded %s' % uri)
    return True
```

3.2 Unit-tests of various S3 functions

```
TEST_DATA_DIR = os.path.dirname(__file__) + '/data/s3'
TEST_BUCKET_NAME = 'test_s3_mock_bucket'
from moto import mock_s3

def get_local_files():
    file_info = []
    for root, dirs, files in os.walk(TEST_DATA_DIR):
        for file_name in files:
            local_path = os.path.join(root, file_name)
            s3_path = os.path.relpath(local_path, TEST_DATA_DIR)
            file_info.append(
                {
                    'local_path': local_path,
                    's3_path': s3_path
                }
            )

    return file_info


@mock_s3
def test_create_bucket():
    s3_client = client('s3')
    create_bucket = s3_client.create_bucket(Bucket=TEST_BUCKET_NAME)
    assert create_bucket['ResponseMetadata']['HTTPStatusCode'] == 200
    return TEST_BUCKET_NAME

@mock_s3
def test_blob_upload_file():
    test_create_bucket()
    file_info = get_local_files()
    file = random.choice(file_info)
    assert blob.upload_file(file['local_path'], TEST_BUCKET_NAME,
                            file['s3_path'])

@mock_s3
def test_blob_object_exists():
    test_create_bucket()
    test_blob_upload_objects()
    response = blob.object_exists(TEST_BUCKET_NAME,
                                  'some_path',
                                  modified_since=datetime.now(),
                                  return_response=True,
                                  raise_exception=False)
    assert not response

@mock_s3
def test_blob_upload_objects():
    test_create_bucket()
    file_info = get_local_files()
    for file in file_info:
        with open(file['local_path'], 'r') as f:
            file_content = f.read()
        assert blob.upload_object(TEST_BUCKET_NAME, file['s3_path'],
                                  file_content)


@mock_s3
def test_blob_upload_object_exception():
    test_create_bucket()
    file_info = get_local_files()
    file = random.choice(file_info)
    with open(file['local_path'], 'r') as f:
        file_content = f.read()
    with pytest.raises(Exception) as e:
        blob.upload_object('FAKE_BUCKET', file['s3_path'], file_content,
                           raise_exception=True)
    assert 'An error occurred (NoSuchBucket)' in e.value.message


@mock_s3
def test_blob_list_objects():
    test_create_bucket()
    test_blob_upload_objects()
    file_info = get_local_files()

    s3_all_paths = sorted(item['s3_path'] for item in file_info)
    s3_list_all_objects = blob.list_objects(TEST_BUCKET_NAME, '', '')
    assert s3_all_paths == sorted(s3_list_all_objects)

    path_xyz = 'xyz'
    s3_paths_xyz = sorted([item for item in s3_all_paths
                           if item.startswith(path_xyz)])
    s3_list_objects_xyz = blob.list_objects(TEST_BUCKET_NAME, path_xyz, '')
    assert s3_paths_xyz == sorted(s3_list_objects_xyz)

    path_some_path = 'some_path'
    s3_paths_some_path = sorted([item for item in s3_all_paths
                                 if item.startswith(path_some_path)])
    s3_list_objects_some_path = blob.list_objects(
        TEST_BUCKET_NAME,
        path_some_path,
        ''
    )
    assert s3_paths_some_path == sorted(s3_list_objects_some_path)

    limit = 2
    s3_list_objects_limit = blob.list_objects(
        TEST_BUCKET_NAME,
        '',
        '',
        limit
    )
    assert s3_all_paths[:limit] == sorted(s3_list_objects_limit)

    s3_paths_strip_prefix = sorted([item[len(path_some_path):]
                                    for item in s3_paths_some_path])

    s3_list_objects_strip_prefix = blob.list_objects(
        TEST_BUCKET_NAME,
        path_some_path,
        '',
        strip_prefix=True
    )
    assert s3_paths_strip_prefix == sorted(s3_list_objects_strip_prefix)

    # test prefixes
    (keys, prefixes) = blob.list_objects(
        TEST_BUCKET_NAME,
        '',
        return_prefixes=True
    )
    assert keys == ['a.json', 'c.yaml', 'd.csv']
    assert prefixes == ['manual/', 'some_path/']


@mock_s3
def test_blob_download_file():
    test_create_bucket()
    test_blob_upload_objects()
    file_info = get_local_files()
    file = random.choice(file_info)
    local_path = '/tmp/' + file['s3_path'].split('/')[-1]
    assert blob.download_file(TEST_BUCKET_NAME, file['s3_path'], local_path)
    file_exists = os.path.isfile(local_path)
    assert file_exists
    if file_exists:
        os.remove(local_path)
```

4 PATCHING

4.1 The Patch Context Manager

Monkey patching is easy. Un-monkey patching is hard, patch does it for you. Replacing one object with other and store it for you.

```
>>> import module

>>> from mock import patch

>>> original = module.SomeObject

>>> with patch('module.SomeObject') as mock_obj:

...     assert module.SomeObject is mock_obj

....

>>> assert module.SomeObject is original
```

4.2 The Patch Decorator

Patch can also be used as a decorator, often to decorate test methods. You can nest them for multiple patches. The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

```
class TestSomething(unittest2.TestCase):

  @patch('module.SomeObject')

  @patch('module.OtherObject')

  def test_something(Self, mock_other_object,
mock_Some_object):

    pass
```

patch is done where look up happened, not necessarily where object is defined. Patch() is straightforward to use. The key is to do the patching in the right namespace. How would you test SomeClass.method without instantiating BigExpensiveClass?

```
from othermodule import BigClass

class SomeClass(object):

   def method(self, arg):

     self.thing = BigClass(arg)

from mock import patch,sentinel

from module import SomeClass

with patch('module.BigClass') as MockClass:

  Obj = SomeClass()

  Arg = sentinel.arg

  Obj.method(arg)

  instance = MockClass.return_value

  assert obj.thing is instance

  MockClass.assert_called_once_with(arg)
```

Mocking module of bigclass and look up is happening in our module.

What if code look like below shown: It means lookup is happening in another module not inside our namespace

```
What if the code looked like this instead?
import othermodule


class SomeClass(object):


  def method(Self, arg):


    self.thing = othermodule.BigClass(arg)

from mock import patch,sentinel

from module import SomeClass

with patch('othermodule.BigClass') as MockClass:

  Obj = SomeClass()

  Arg = sentinel.arg

  Obj.method(arg)

  instance = MockClass.return_value

  assert obj.thing is instance

  MockClass.assert_called_once_with(arg)
```

5 AWS AND GOOGLE STORAGE FUNCTION'S TEST

5.1 AWS Functions

5.1.1 Functions(utils.py)

```
def get_env_name():

  env = os.environ.get('ENV')

  if env is None:

    env = os.environ.get('AWS_DEFAULT_PROFILE')

    if env is not None:

      env = env.replace('core-', '')

    else:

      raise Exception('get_env_name: unable to get ENV '
+

                'or AWS_DEFAULT_PROFILE')

  return env



def get_aws_region():

  aws_region = os.environ.get('AWS_DEFAULT_REGION')

  if aws_region is None:

    aws_region = os.environ.get('AWS_REGION')

  if aws_region is None:

    raise Exception('get_aws_region: unable to get
AWS_DEFAULT_REGION')

  return aws_region
```

```
def get_aws_account_id():

  aws_account_id = os.environ.get('AWS_ACCOUNT_ID')

  if aws_account_id is None:

    raise Exception('get_aws_account_id: unable to get
AWS_ACCOUNT_ID')

  return aws_account_id


def to_bool(value):

  if not value:

    return False

  elif isinstance(value, bool):

    return value

  elif isinstance(value, basestring):

    return bool(strtobool(value.lower()))

  else:

    return bool(value)


def bytes_size(b):

  if b == 0:

    return "0B"

  size_name = ("B", "KB", "MB", "GB", "TB", "PB", "EB",
"ZB", "YB")

  i = int(math.floor(math.log(b, 1024)))

  p = math.pow(1024, i)

  s = round(b / p, 2)

  return "%s %s" % (s, size_name[i])
```

```
def fatal(msg):

  logging.critical(msg)

  if flask.has_request_context():

    abort(500)

  else:

    sys.exit(1)
```

5.1.2 Tests

```
import utils

import os

import pytest

import sys

import time


ENV = 'unittest'

BUCKET = 'test-%s' % ENV

os.environ['ENV'] = ENV


def test_get_env_name():

  for x in ['AWS_DEFAULT_PROFILE', 'ENV']:

    os.environ.pop(x, None)

  with pytest.raises(Exception) as e:

    utils.get_env_name()

  assert 'unable to get ENV' in e.value.message

  os.environ['AWS_DEFAULT_PROFILE'] = 'core-' + ENV

  assert utils.get_env_name() == ENV
```

```
    os.environ.pop('AWS_DEFAULT_PROFILE', None)

    os.environ['ENV'] = ENV

    assert utils.get_env_name() == ENV


def test_get_aws_region():

    for x in ['AWS_DEFAULT_REGION', 'AWS_REGION']:

        os.environ.pop(x, None)

    with pytest.raises(Exception) as e:

        utils.get_aws_region()

    assert 'unable to get AWS_DEFAULT_REGION' in
e.value.message


    os.environ['AWS_REGION'] = 'us-east-1'

    assert utils.get_aws_region() == 'us-east-1'


def test_get_aws_account_id():

    os.environ['AWS_ACCOUNT_ID'] = 'TEST ID'

    assert utils.get_aws_account_id() == 'TEST ID'

    os.environ.pop('AWS_ACCOUNT_ID', None)

    with pytest.raises(Exception) as e:

        utils.get_aws_account_id()

    assert 'unable to get AWS_ACCOUNT_ID' in
e.value.message


def test_to_bool():

    assert utils.to_bool(None) is False

    assert utils.to_bool(True) is True

    assert utils.to_bool('False') is False

    assert utils.to_bool(1) is True


def test_bytes_size():

    assert utils.bytes_size(0) == '0B'

    assert utils.bytes_size(1024) == '1.0 KB'


def test_fatal():

    with pytest.raises(SystemExit) as e:

        utils.fatal('foo')

    assert e.type == SystemExit

    assert e.value.code == 1
```

5.2 For GOOGLE STORAGE Cloud Functions

Unit tests for uploading file to google storage using python:

```
class TestDate(unittest.TestCase):

    def test_upload_to_Google_Storage(self):

        from google.cloud import storage


        blob_mock = Mock(spec=storage.Blob)

        bucket_mock = Mock(spec=storage.Bucket)

        bucket_mock.blob.return_value = blob_mock


        storage_client_mock = Mock(spec=storage.Client)

        storage_client_mock.get_bucket.return_value =
bucket_mock

        blob_mock.upload_from_filename = Mock()

        os.path.isfile = Mock()

        os.remove = Mock()

        # mockobject is passed as file name
    response =
upload_to_google_storage.upload_to_GS('test-bucket',
Mock(),

                                      blob_mock,

                                      storage_client_mock)

    self.assertTrue(blob_mock.exists())

    self.assertIsNotNone(response)
```

```
class TestBlob(unittest.TestCase):

  def test_list_blobs(self):

    from google.cloud import storage



    storage_client_mock =
MagicMock(spec=storage.Client)

    bucket_mock = MagicMock(spec=storage.Bucket)

    storage_client_mock.get_bucket.return_value =
bucket_mock

    iterator = blob.list_blobs(storage_client_mock, 'test-
bucket')

    blobs = list(iterator)

    self.assertEqual(blobs, [])



if __name__ == '__main__':

  unittest.main()
```

6 CONCLUSION

Mocks are objects that register calls they receive. In the test assertion, we can verify on Mocks that all expected actions were performed. We use mocks when we don't want to invoke production code or when there is no easy way to verify, that intended code was executed. There is no return value and no easy way to check system state change. An example can be a functionality that calls e-mail sending service. We don't want to send e-mails each time we run a test. Moreover, it is not easy to verify in tests that a right email was sent. The only thing we can do is to verify the outputs of the functionality that is exercised in our test. In other words, verify that e-mail sending service was called. In addition to this, moto which is a library, allows your tests to easily mock out AWS Services. It creates a full, blank environment. Moto also comes with a stand-alone server allowing you to mock out an AWS HTTP endpoint. Moto provides some internal APIs to view and change the state of the backends, This API resets the state of all of the backend. Moto can be used in any language such as Ruby, Java, Javascript, Python.

REFERENCES

[1] https://docs.python.org/3/library/unittest.mock.html
[2] https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
[3] https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da
[4] https://medium.com/adobetech/effectively-testing-our-aws-s3-utilization-using-the-s3mock-f139ebf81572

BIOGRAPHIES

I am having total 3 years of experience with Accenture. I opted for Electronics and Electrical engineering as my undergraduate specialization. I am also a food blogger as well as solo traveler. I was All India finalist for beauty pageant.