**patch()**

unittest.mock provides a powerful mechanism for mocking objects, called [patch()](https://docs.python.org/3/library/unittest.mock.html#patch), which looks up an object in a given module and replaces that object with a Mock.

Usually, you use patch() as a decorator or a context manager to provide a scope in which you will mock the target object.

patch() returns an instance of [MagicMock](https://docs.python.org/3/library/unittest.mock.html" \l "unittest.mock.MagicMock), which is a Mock subclass. MagicMock is useful because it implements most [magic methods](https://dbader.org/blog/python-dunder-methods) for you, such as .\_\_len\_\_(), .\_\_str\_\_(), and .\_\_iter\_\_(), with reasonable defaults.

### **patch() as a Decorator**

If you want to mock an object for the duration of your entire test function, you can use patch() as a function [decorator](https://realpython.com/primer-on-python-decorators/).

When a function is decorated using @patch, a mock of the class or a method or a function passed as the target to @patch is returned and passed as an argument to the decorated function.

### **Example:**

### **patch() as a Context Manager**

Sometimes, you’ll want to use patch() as a [context manager](https://dbader.org/blog/python-context-managers-and-with-statement) rather than a decorator. Some reasons why you might prefer a context manager include the following:

* You only want to mock an object for a part of the test scope.
* You are already using too many decorators or parameters, which hurts your test’s readability.

To use patch() as a context manager, you use Python’s with statement:

### **Patching an Object’s Attributes:**

Let’s say you only want to mock one method of an object instead of the entire object. You can do so by using [patch.object()](https://docs.python.org/3/library/unittest.mock.html#patch-object).

All the above-mentioned details are captured in the example below:

import unittest  
from unittest import TestCase  
from unittest.mock import patch  
from UnitTestingInPython.my\_blog import Blog  
from UnitTestingInPython.my\_blog import requests  
  
class OTFError(Exception):  
  
 # Constructor or Initializer  
 def \_\_init\_\_(self, value, code='10'):  
 self.code = code  
 self.value = value  
  
 # \_\_str\_\_ is to print() the value  
  
 def \_\_str\_\_(self):  
 return (repr(self.value + "With code value as "+self.code))  
  
  
  
class TestBlog(TestCase):  
 # Test how posts will behave when there is a request timeout in requests.get()  
 def setUp(self):  
 self.blog = Blog("sabya")  
  
 def throws\_exception(self, \*args, \*\*kwargs):  
 print("Throws Exception")  
 raise OTFError("OTF timeout exception")  
  
 @patch('UnitTestingInPython.my\_blog.requests', autospec=True)  
 def test\_posts\_timeout\_with\_decorator(self, mock\_requests):  
 # Here the patch returns a MagicMock() which is passed as an argument to  
 # test\_posts\_timeout\_with\_decorator as mock\_requests  
 # We can set .return\_value and .side\_effect on a Mock directly.  
  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* test\_posts\_timeout\_with\_decorator \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 mock\_requests.get.side\_effect = self.throws\_exception  
 ret = None  
 ret = self.blog.posts()  
  
 def test\_posts\_timeout\_with\_context\_manager(self):  
 print(" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* test\_posts\_timeout\_with\_context\_manager \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 with patch('UnitTestingInPython.my\_blog.requests') as mock\_requests:  
 mock\_requests.get.side\_effect = self.throws\_exception  
 ret = self.blog.posts()  
 self.assertIsInstance(ret, str)  
 self.assertIsNotNone(ret)  
  
 @patch.object(requests, 'get', side\_effect=requests.exceptions.Timeout)  
 def test\_posts\_timeout\_with\_object(self, mock\_requests):  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\* Testing with objects \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 with self.assertRaises(requests.exceptions.Timeout):  
 ret = self.blog.posts\_timeout()  
 self.assertIsInstance(ret, requests.exceptions.Timeout)  
 self.blog.posts\_timeout.assert\_called\_once\_with()  
 self.assertIsNotNone(ret)  
  
 #This is an example for return\_value setting using mock object  
 @patch('UnitTestingInPython.my\_blog.Blog')  
 def test\_blog\_posts\_using\_return\_value(self, mock\_blog):  
 blog = mock\_blog()  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\* test\_blog\_posts\_using\_return\_value \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 blog.posts.return\_value = [  
 {  
 'userId': 1,  
 'id': 1,  
 'title': 'Test Title',  
 'body': 'Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy\ lies a small unregarded yellow sun.'  
 }  
 ]  
 response = blog.posts()  
 self.assertIsNotNone(response)  
  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

where my\_blog.py is :

import requests  
  
class Blog:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def posts\_timeout(self):  
 print("Posts is called")  
 try:  
 response = requests.get("https://jsonplaceholder.typicode.com/posts")  
 return response.json()  
 except:  
 raise requests.exceptions.Timeout  
  
  
 def posts(self):  
 print("Posts is called")  
 try:  
 response = requests.get("https://jsonplaceholder.typicode.com/posts")  
 return response.json()  
 except:  
 return "Some exception received"  
  
  
 def \_\_repr\_\_(self):  
 return '<Blog: {}>'.format(self.name)