Python patch()

Summary: in this tutorial, you'll learn how to use the Python `patch()` to replace a target with a mock object temporarily.

Introduction to the Python patch

The `unittest.mock` module has a `patch()` that allows you to temporarily replace a target with a mock object.

A target can be a function (https://www.pythontutorial.net/python-basics/python-functions/) , a method (https://www.pythontutorial.net/python-oop/python-methods/) , or a class (https://www.pythontutorial.net/python-oop/python-class/) . It's a string with the following format:

```
'package.module.className'
```

To use the `patch()` correctly, you need to understand two important steps:

- Identify the target
- How to call `patch()`

Identifying the target

To identify a target:

- The target must be importable.
- And patch the target where it is used, not where it comes from.

Calling patch

Python provides you with three ways to call `patch()` :

- Decorators for a function or a class.
- Context manager
- Manual start/stop

When you use the `patch()` as a decorator of a function or class, inside the function or class the target is replaced with a new object.

If you use the patch in a context manager, inside the `with` statement, the target is replaced with a new object.

In both cases, when the function or the `with` statement exits, the patch is undone.

Python patch examples

Let's create a new module called `total.py` for demonstration purposes:

```
def read(filename):
    """ read a text file and return a list of numbers """
    with open(filename) as f:
        lines = f.readlines()
        return [float(line.strip()) for line in lines]


def calculate_total(filename):
    """ return the sum of numbers in a text file """
    numbers = read(filename)
    return sum(numbers)
```

How it works.

The `read()` function reads a text file, converts each line into a number, and returns a list of numbers. For example, a text file has the following lines:

```
1
2
3
```

the `read()` function will return the following list:

```
[1, 2, 3]
```

The `calculate_total()` function uses the `read()` function to get a list of numbers from a file and returns the sum of the numbers.

To test `calculate_total()`, you can create a `test_total_mock.py` module and mock the `read()` function as follows:

```
import unittest

from unittest.mock import MagicMock

import total


class TestTotal(unittest.TestCase):
    def test_calculate_total(self):
        total.read = MagicMock()
        total.read.return_value = [1, 2, 3]
        result = total.calculate_total('')
        self.assertEqual(result, 6)
```

Run the test:

```
python -m unittest test_total_mock.py -v
```

Output:

```
test_calculate_total (test_total_mock.TestTotal) ... ok


----------------------------------------------------------------------
Ran 1 test in 0.001s


OK
```

Instead of using the `MagicMock()` object directly, you can use the `patch()`.

1) Using patch() as a decorator

The following test module `test_total_with_patch_decorator.py` tests the `total.py` module using the `patch()` as a function decorator:

```
import unittest
from unittest.mock import patch
import total



class TestTotal(unittest.TestCase):
    @patch('total.read')
    def test_calculate_total(self, mock_read):
        mock_read.return_value = [1, 2, 3]
        result = total.calculate_total('')
        self.assertEqual(result, 6)
```

How it works.

First, import the patch from the `unittest.mock` module:

```
from unittest.mock import patch
```

Second, decorate the `test_calculate_total()` test method with the `@patch` decorator. The target is the read function of the total module.

```
@patch('total.read')
def test_calculate_total(self, mock_read):
    # ...
```

Because of the `@patch` decorator, the `test_calculate_total()` method has an additional argument mock_read which is an instance of the MagicMock.

Note that you can name the parameter whatever you want.

Inside the `test_calculate_total()` method, the `patch()` will replace the total.`read()` function with the mock_read object.

Third, assign a list to the return_value of the mock object:

```
 mock_read.return_value = [1, 2, 3]
```

Finally, call the `calculate_total()` function and use the `assertEqual()` method to test if the total is 6.

Because the mock_read object will be called instead of the total.`read()` function, you can pass any filename to the `calculate_total()` function:

```
result = total.calculate_total('')
self.assertEqual(result, 6)
```

Run the test:

```
python -m unittest test_total_patch_decorator -v
```

Output:

```
test_calculate_total (test_total_patch_decorator.TestTotal) ... ok


----------------------------------------------------------------------
Ran 1 test in 0.001s
```

```
OK
```

2) Using patch() as a context manager

The following example illustrates how to use the `patch()` as a context manager:

```
import unittest
from unittest.mock import patch
import total


class TestTotal(unittest.TestCase):
    def test_calculate_total(self):
        with patch('total.read') as mock_read:
            mock_read.return_value = [1, 2, 3]
            result = total.calculate_total('')
            self.assertEqual(result, 6)
```

How it works.

First, patch `total.read()` function using as the `mock_read` object in a context manager:

```
with patch('total.read') as mock_read:
```

It means that within the `with` block, the `patch()` replaces the `total.read()` function with the mock_read object.

Second, assign a list of numbers to the `return_value` property of the `mock_read` object:

```
mock_read.return_value = [1, 2, 3]
```

Third, call the `calculate_total()` function and test if the result of the `calculate_total()` function is equal 6 using the `assertEqual()` method:

```
result = total.calculate_total('')
self.assertEqual(result, 6)
```

Run the test:

```
python -m unittest test_total_patch_ctx_mgr -v
```

Output:

```
test_calculate_total (test_total_patch_ctx_mgr.TestTotal) ... ok

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

Ran 1 test in 0.001s


OK
```

3) Using patch() manually

The following test module (`test_total_patch_manual.py`) shows how to use `patch()` manually:

```
import unittest
from unittest.mock import patch
import total



class TestTotal(unittest.TestCase):
    def test_calculate_total(self):
        # start patching
        patcher = patch('total.read')

        # create a mock object
        mock_read = patcher.start()

        # assign the return value
        mock_read.return_value = [1, 2, 3]
        # test the calculate_total
        result = total.calculate_total('')
        self.assertEqual(result, 6)

        # stop patching
        patcher.stop()
```

How it works.

First, start a patch by calling `patch()` with a target is the `read()` function of the `total` module:

```
patcher = patch('total.read')
```

Next, create a mock object for the `read()` function:

```
mock_read = patcher.start()
```

Then, assign a list of numbers to the `return_value` of the `mock_read` object:

```
result = total.calculate_total('')
self.assertEqual(result, 6)
```

After that, call the `calculate_total()` and test its result.

```
def test_calculate_total(self):
    self.mock_read.return_value = [1, 2, 3]
    result = total.calculate_total('')
    self.assertEqual(result, 6)
```

Finally, stop patching by calling the `stop()` method of the patcher object:

```
patcher.stop()
```

Summary

- Use the `patch()` from `unittest.mock` module to temporarily replace a target with a mock object.
- Use the `patch()` as a decorator, a context manager, or manually call `start()` and `stop()` patching.