Python Stubs

Summary: in this tutorial, you'll learn how to use Python stubs to isolate parts of your program from each other for unit testing.

Introduction to the Python stubs

Stubs are test doubles that return hard-coded values. The primary purpose of stubs is to prepare a specific state of the system under test.

Stubs are beneficial because they return consistent results, making the test easier to write. Also, you can run tests even if the components that stubs are present are not working yet.

Suppose you need to develop an alarm system that monitors the temperature of a room like a server room.

To do that you need to set up a temperature sensor device and use the data from that sensor to alert if the temperature is below or above a specific temperature.

First, define a `Sensor` class in the `sensor.py` module:

```
import random
```

```
class Sensor:
    @property
    def temperature(self):
        return random.randint(10, 45)
```

The `Sensor` class has a temperature property that returns a random temperature between 10 and 45. In the real world, the `Sensor` class needs to connect to the sensor device to get the actual temperature.

Second, define the `Alarm` class that uses a `Sensor` object:

```
from sensor import Sensor


class Alarm:
    def __init__(self, sensor=None) -> None:
        self._low = 18
        self._high = 24
        self._sensor = sensor or Sensor()
        self._is_on = False

    def check(self):
        temperature = self._sensor.temperature
        if temperature < self._low or temperature > self._high:
            self._is_on = True

    @property
    def is_on(self):
        return self._is_on
```

By default, the `is_on` property of the `Alarm` is off (`False`). The `check()` method turns the alarm on if the temperature is lower than 18 or higher than 42.

Once the `is_on` property of an `Alarm` object is on, you can send it to the alarm device to alert accordingly.

Because the `temperature()` method of the `Sensor` returns a random temperature, it'll be difficult to test various scenarios to ensure the `Alarm` class works properly.

To resolve it, you can define a stub for the `Sensor` class called `TestSensor`. The `TestSensor` has the `temperature` property that returns a value provided when its object is initialized.

Third, define the `TestSensor` in `test_sensor.py` module:

```
class TestSensor:
    def __init__(self, temperature) -> None:
        self._temperature = temperature

    @property
    def temperature(self):
        return self._temperature
```

The `TestSensor` class is like the `Sensor` class except that the `temperature` property returns a value specified in the constructor.

Fourth, define a `TestAlarm` class in the `test_alarm.py` test module and import the `Alarm` and `TestSensor` from the `alarm.py` and `sensor.py` modules:

```
import unittest
from alarm import Alarm
from test_sensor import TestSensor


class TestAlarm(unittest.TestCase):
   pass
```

Fifth, test if the alarm is off by default:

```
import unittest
from alarm import Alarm
from test_sensor import TestSensor
```

```
class TestAlarm(unittest.TestCase):
    def test_is_alarm_off_by_default(self):
        alarm = Alarm()
        self.assertFalse(alarm.is_on)
```

In the `test_is_alarm_off_by_default` we create a new alarm instance and use the `assertFalse()` method to check if the `is_on` property of the alarm object is `False`.

Run the test:

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

Output:

```
test_is_alarm_off_by_default (test_alarm.TestAlarm) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Sixth, test the `check()` method of the `Alarm` class in case the temperature is too high:

```
import unittest
from alarm import Alarm
from test_sensor import TestSensor


class TestAlarm(unittest.TestCase):
    def test_is_alarm_off_by_default(self):
        alarm = Alarm()
        self.assertFalse(alarm.is_on)

    def test_check_temperature_too_high(self):
        alarm = Alarm(TestSensor(25))
```

```
        alarm.check()
        self.assertTrue(alarm.is_on)
```

In the `test_check_temperature_too_high()` test method:

- Create an instance of the `TestSensor` with temperature 25 and passes it to the `Alarm` constructor.
- Call the `check()` method of the alarm object.
- Use the `assertTrue()` to test if the `is_on` property of the alarm is `True`.

Run the test:

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

Output:

```
test_check_temperature_too_high (test_alarm.TestAlarm) ... ok
test_is_alarm_off_by_default (test_alarm.TestAlarm) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

The alarm is on because the temperature is higher than 24.

Seventh, test the `check()` method of the Alarm class when the temperature is too low:

```
import unittest
from alarm import Alarm
from test_sensor import TestSensor


class TestAlarm(unittest.TestCase):
    def test_is_alarm_off_by_default(self):
        alarm = Alarm()
```

```
        self.assertFalse(alarm.is_on)

    def test_check_temperature_too_high(self):
        alarm = Alarm(TestSensor(25))
        alarm.check()
        self.assertTrue(alarm.is_on)

    def test_check_temperature_too_low(self):
        alarm = Alarm(TestSensor(17))
        alarm.check()
        self.assertTrue(alarm.is_on)
```

In the `test_check_temperature_too_low()` test method:

- Create an instance of the `TestSensor` with temperature 17 and passes it to the Alarm constructor.
- Call the `check()` method of the alarm object.
- Use the `assertTrue()` to test if the is_on property of the alarm is True.

Run the test:

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

Output:

```
test_check_temperature_too_high (test_alarm.TestAlarm) ... ok
test_check_temperature_too_low (test_alarm.TestAlarm) ... ok
test_is_alarm_off_by_default (test_alarm.TestAlarm) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Seventh, test the `check()` method of the `Alarm` class if the temperature is in the safe range (18, 24):

```
import unittest
from alarm import Alarm
from test_sensor import TestSensor


class TestAlarm(unittest.TestCase):
    def test_is_alarm_off_by_default(self):
        alarm = Alarm()
        self.assertFalse(alarm.is_on)

    def test_check_temperature_too_high(self):
        alarm = Alarm(TestSensor(25))
        alarm.check()
        self.assertTrue(alarm.is_on)

    def test_check_temperature_too_low(self):
        alarm = Alarm(TestSensor(15))
        alarm.check()
        self.assertTrue(alarm.is_on)

    def test_check_normal_temperature(self):
        alarm = Alarm(TestSensor(20))
        alarm.check()
        self.assertFalse(alarm.is_on)
```

In the `test_check_normal_temperature()` method we create a TestSensor with the temperature 20 and pass it to the Alarm constructor. Since the temperature is in the range (18, 24), the alarm should be off.

Run the test:

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

Output:

```
test_check_normal_temperature (test_alarm.TestAlarm) ... ok
test_check_temperature_too_high (test_alarm.TestAlarm) ... ok
test_check_temperature_too_low (test_alarm.TestAlarm) ... ok
test_is_alarm_off_by_default (test_alarm.TestAlarm) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

Using MagicMock class to create stubs

Python provides you with the `MagicMock` object in the `unittest.mock` (https://www.pythontutorial.net/python-unit-testing/python-unittest-mock/) module that allows you to create stubs more easily.

To create a stub for the `Sensor` class using the `MagicMock` class, you pass the `Sensor` class to the `MagicMock()` constructor:

```
mock_sensor = MagicMock(Sensor)
```

The `mock_sensor` is the new instance of the `MagicMock` class that mocks the `Sensor` class.

By using the `mock_sensor` object, you can set its property or call a method. For example, you can assign a specific temperature e.g., 25 to the `temperature` property of the mock sensor like this:

```
mock_sensor.temperature = 25
```

The following shows the new version of the `TestAlarm` that uses the `MagicMock` class:

```
import unittest
from unittest.mock import MagicMock
from alarm import Alarm
from sensor import Sensor
```

```
class TestAlarm(unittest.TestCase):
    def setUp(self):
        self.mock_sensor = MagicMock(Sensor)
        self.alarm = Alarm(self.mock_sensor)

    def test_is_alarm_off_by_default(self):
        alarm = Alarm()
        self.assertFalse(alarm.is_on)

    def test_check_temperature_too_high(self):
        self.mock_sensor.temperature = 25
        self.alarm.check()
        self.assertTrue(self.alarm.is_on)

    def test_check_temperature_too_low(self):
        self.mock_sensor.temperature = 15
        self.alarm.check()
        self.assertTrue(self.alarm.is_on)

    def test_check_normal_temperature(self):
        self.mock_sensor.temperature = 20
        self.alarm.check()
        self.assertFalse(self.alarm.is_on)
```

Using patch() method

To make it easier to work with `MagicMock`, you can use the `patch()` (https://www.pythontutorial.net/python-unit-testing/python-patch/) as a decorator. For example:

```
import unittest
from unittest.mock import patch
from alarm import Alarm
```

```
class TestAlarm(unittest.TestCase):
    @patch('sensor.Sensor')
    def test_check_temperature_too_low(self, sensor):
        sensor.temperature = 10
        alarm = Alarm(sensor)
        alarm.check()
        self.assertTrue(alarm.is_on)
```

In this example, we use a `@patch` decorator on the `test_check_temperature_too_low()` method. In the decorator, we pass the `sensor.Sensor` as a target to patch.

Once we use the `@patch` decorator, the test method will have the second parameter which is an instance of the `MagicMock` that mocks the `sensor.Sensor` class.

Inside the test method, we set the temperature property of the sensor to 10, create a new instance of the Alarm class, and call `check()` method, and use the `assertTrue()` method to test if the alarm is on.

Run the test:

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

Output:

```
test_check_temperature_too_low (test_alarm_with_patch.TestAlarm) ... ok

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

Summary

- Use stubs to return hard-coded values for testing.
- Use `MagicMock` class of `unittest.mock` module to create stubs.
- Use `patch()` to create `MagicMock` more easily.