Introducción al testeo con Mocks

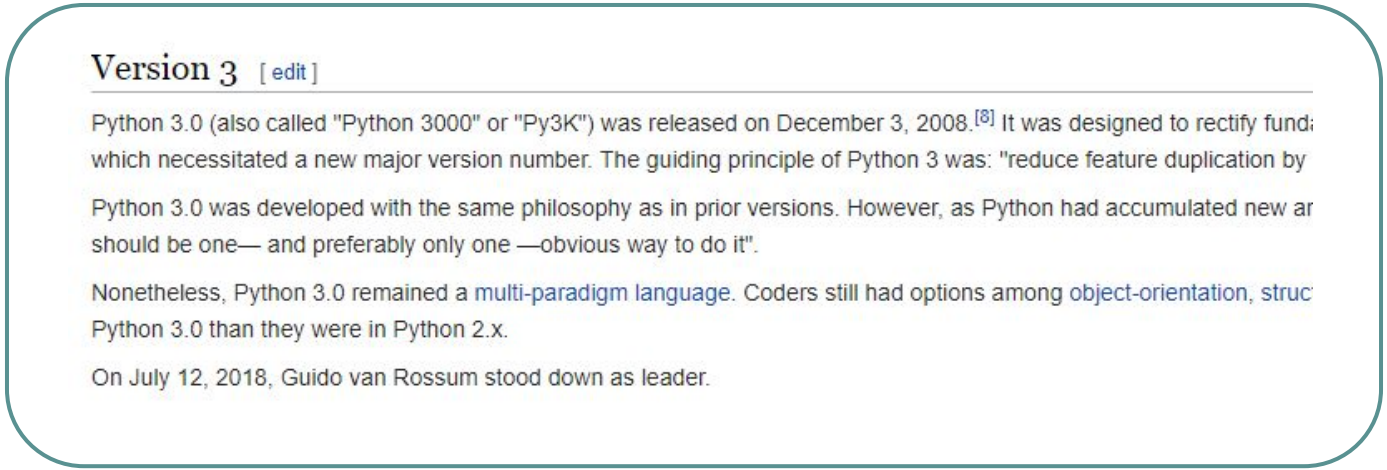
Ester Ramos Carmona

Antes de empezar

Pausa para la publicidad

Python 2 deja de tener soporte en 2020.

Venid con nosotros a esta nueva* era de Python 3.



Version 3 [edit]

Python 3.0 (also called "Python 3000" or "Py3K") was released on December 3, 2008.[8] It was designed to rectify fund
which necessitated a new major version number. The guiding principle of Python 3 was: "reduce feature duplication by

Python 3.0 was developed with the same philosophy as in prior versions. However, as Python had accumulated new ar
should be one— and preferably only one —obvious way to do it".

Nonetheless, Python 3.0 remained a multi-paradigm language. Coders still had options among object-orientation, struc
Python 3.0 than they were in Python 2.x.

On July 12, 2018, Guido van Rossum stood down as leader.

*La palabra “nueva” puede, en realidad, significar lo opuesto

Pausa para la publicidad

La evolución de los dinosaurios: migrando de Python 2 a Python 3

Por Marta Gómez Macías

¿Por qué usar Mocks?

¿Imposible de testear?

```
1 def creador_de_archivo(nombre):
2   with open(nombre, "w") as f:
3     f.write("Hola, me llamo Ester")
```

¿Imposible de testear?

```
1 def contenido_web(url):
2     response = request.get(url)
3     return response.content
```

¿Qué son los Mocks?

Creando dobles

Un mock crea un objeto que parece igual al original pero nosotros controlamos lo que hace.

```
import unittest.mock
```

Controlando el comportamiento

```
1 def programa(parametros, canal_slack):
2    ...
3   canal_slack(resultado)
4    ...
```

```
from unittest.mock import Mock
```

Controlando el comportamiento

```
1 def programa(parametros, canal_slack):
2    ...
3    canal_slack(resultado)
4    ...
```

```
doble = Mock(return_value="ÉXITO")
```

```
from unittest.mock import Mock
```

Controlando el comportamiento

```
1 def programa(parametros, canal_slack):
2    ...
3    canal_slack(resultado)
4    ...
```

```
doble = Mock(return_value="ÉXITO")
```

```
from unittest.mock import Mock
```

Controlando el comportamiento

```
1 def programa(parametros, canal_slack):
2    ...
3    canal_slack(resultado)
4    ...
```

```
doble = Mock(return_value="ÉXITO")
```



```
programa(parametros, doble)
```

```
from unittest.mock import Mock
```

Controlando el comportamiento

```
1 def programa(parametros, canal_slack):
2    ...
3   canal_slack(resultado)
4    ...
```

```
doble = Mock(return_value="ÉXITO")
```



```
programa(parametros, doble)
```

```
from unittest.mock import Mock
```

Si el objeto está dentro de la función

```
1 def programa(parametros, [redacted]):
2     ...
3     canal_slack(resultado)
4     ...
```

Patch es tu amigo

Si el objeto está dentro de la función

Cambia tu objeto por un mock.

Por defecto este nuevo objeto será un MagicMock.

Muy Importante: El objetivo debe ser importable desde donde estás llamando a patch.

```
unittest.mock.patch('paquete.modulo.objetivo')
```

Patch es tu amigo

Si el objeto está dentro de la función

```
def test():
  assert programa.funcion() == 3
```

Patch es tu amigo

Si el objeto está dentro de la función

```
@patch("path.objeto")
def test():
  assert programa.funcion() == 3
```

Patch es tu amigo

Si el objeto está dentro de la función

```
@patch("path.objeto")
def test(mock_objeto):
  assert programa.funcion() == 3
```

Patch es tu amigo

Si el objeto está dentro de la función

```
@patch("path.objeto")
def test(mock_objeto):
    mock_objeto.return_value = 3
    assert programa.funcion() == 3
```

```python
# modulo.py

def modulo():
    ...
```

```python
# programa.py

from modulo import modulo

def funcion():
  return modulo()
```

```python
# test.py

import programa

@patch("-------------")
def test_funcion(mock_modulo):
  mock.modulo.return_value = 3
  assert programa.funcion() == 3
```

```python
# modulo.py

def m...lo():
    ...
```

Patch

```python
# programa.py

from modulo import modulo

def funcion():
  return modulo()
```

```python
# test.py

import programa

@patch("modulo.modulo")
def test_funcion(mock_modulo):
  mock.modulo.return_value = 3
  assert programa.funcion() == 3
```

```python
# modulo.py

def modulo():
    ...
```

```python
# programa.py

from modulo import m      o

def funcion():
  return modulo()
```

Patch

```python
# test.py

import programa

@patch("-------------")
def test_funcion(mock_modulo):
  mock.modulo.return_value = 3
  assert programa.funcion() == 3
```

```python
# modulo.py

def modulo():
    ...
```

```python
# programa.py

from modulo import m    o

def funcion():
  return modulo()
```

Patch

```python
# test.py

import programa

@patch("programa.modulo")
def test_funcion(mock_modulo):
  mock.modulo.return_value = 3
  assert programa.funcion() == 3
```

Patch: más modos de uso

Context Manager

```
def test_funcion():
    with patch('__main__.target') as mock_target:
      mock_target.return_value = 3
       ...
```

Side effects

Para testear excepciones. Cuidado con excepciones muy generales.

```
mock = Mock()
mock.side_effect = Exception("¯\_(ツ)_/¯")
mock()
Traceback (most recent call last):
   ...
Exception: ¯\_(ツ)_/¯
```

Side effects

Si la función llama a tu mock varias veces.

```
mock = Mock()
mock.side_effect = [3, 2, 1]
mock()
```

Side effects

Si la función llama a tu mock varias veces.

```
mock = Mock()
mock.side_effect = [3, 2, 1]
mock()
3
```

Side effects

Si la función llama a tu mock varias veces.

```
mock = Mock()
mock.side_effect = [3, 2, 1]
mock()
3
mock()
2
```

Side effects

Si la función llama a tu mock varias veces.

```
mock = Mock()
mock.side_effect = [3, 2, 1]
mock()
3
mock()
2
mock()
1
```

Inyección de dependencias

Es mejor si tu objetivo a mockear está entre las dependencias.

No tienes que saber cómo está implementado el objeto

```
def do_thing(url):
    ...
    response = requests.get(url)
    return response
```

```
def do_thing(url, session):
    ...
    response = session.get(url)
    return response
```

Inyección de dependencias

Es mejor si tu objetivo a mockear está entre las dependencias.

No tienes que saber cómo está implementado el objeto

```
def do_thing(url):
    ...
    response = request.get(url)
    return response
```

Patch

```
def do_thing(url, session):
    ...
    response = session.get(url)
    return response
```

Mock

Inyección de dependencias

Hacer patch implica que tienes que saber lo que hay dentro de la función.

```
def do_thing(url):
    ...
    response = requests.get(url)
    return response
```

Hacer mock de una dependencia te permite no tener que saber lo que está pasando dentro de la función.

```
def do_thing(url, session):
```

Inyección de dependencias

Hacer patch implica que tienes que saber lo que hay dentro de la función.

```
def do_thing(url):
    ...
    response = requests.get(url)
    return response
```

Hacer mock de una dependencia te permite no tener que saber lo que está pasando dentro de la función.

```
def do_thing(url, session):
```

Asserts

También puedes comprobar si tu mock o uno de sus métodos ha sido llamado.

```
mock = Mock()
mock.method()

mock.method.assert_called()
```

Asserts

Comprobar que la llamada sólo se ha hecho una vez.

```
mock = Mock()
mock.method()

mock.method.assert_called_once()
```

Asserts

O que la llamada se ha hecho con unos argumentos específicos.

```
mock = Mock()
mock.method(1, 2, 3)

mock.method.assert_called_with(1, 2, 3)
```

```
from unittest.mock import call
```

Asserts

Si el mock se llama varias veces.

```
mock = Mock()
mock(1)
mock(2)
mock(3)

calls = [call(1), call(2), call(3)]

mock.assert_has_calls(calls)
```

Spec y Autospec

Crear un método y llamar a assert se hacen de la misma manera con Mock.

```
mock = Mock()
mock(1)

mock.assret_called_once_with(1)
```

Spec y Autospec

Usando spec el mock sólo podrá acceder a atributos que existan en la clase real.

```
import os

mock = Mock(spec=os.listdir)
mock(".")

mock.assret_called_with
```

Spec y Autospec

Usando spec el mock sólo podrá acceder a atributos que existan en la clase real.

```
import os

mock = Mock(spec=os.listdir)
mock(".")

mock.assret_called_with

Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

Spec y Autospec

Cuando parcheas puedes usar autospec y lo hará automáticamente

```
import os

with patch("__main__.os", autospec=True):
  print(os.listir())
```

Spec y Autospec

Cuando parcheas puedes usar autospec y lo hará automáticamente

```
import os

with patch("__main__.os", autospec=True):
  print(os.listir())

Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'listir'
```

Mock vs MagicMock

MagicMock te permite usar métodos mágicos sin tener que configurarlos.

Pero a veces quieres que tu objeto devuelva un error si se intenta acceder a él.

Mock vs MagicMock

```
# Ejemplos de metodos magicos

# Slicing
lista[indice]
# Operadores aritmeticos
objeto + otro_objeto
# Comparacion
objeto == otro_objeto

len(objeto)
```

Gracias

¿Preguntas?