Модуль Mock: макеты-пустышки в тестировании

Источник: <https://habr.com/post/141209/>

*Mock* на английском значит «имитация», «подделка». Модуль с таким названием помогает сильно упростить тесты модулей на Питоне.  
  
Принцип его работы простой: если нужно тестировать функцию, то всё, что не относится к ней самой (например, чтение с диска или из сети), можно подменить макетами-пустышками. При этом тестируемые функции не нужно адаптировать для тестов: Mock подменяет объекты в других модулях, даже если код не принимает их в виде параметров. То есть, тестировать можно вообще без адаптации под тесты.  
  
Такое поведение — уже не надувные ракетные установки, а целая надувная земля, вокруг которой могут летать испытуемые ракеты и самолёты.  
  
  
  
  
  
Подменять объекты в других модулях, что называется monkey patch, нам иногда в работе приходится. Но в отличие от ручных патчей, Mock умеет делать очень сложные подстановки, пачками и цепочками вызова, а также убраться за собой, не оставляя побочных эффектов.  
  
Пакет занимает меньше 1 мегабайта и устанавливается очень просто:  
  
$ pip install mock  
или  
$ easy\_install mock  
  
И теперь им можно пользоваться.

Подмена функции

Скажем, наша функция считает что-то, причём очень долго:

from itertools import permutations

def real(name):

if len(name) < 10:

raise ValueError('String too short to calculate statistics.')

y = 0

for i in permutations(xrange(len(name)), 10):

y += i

print y

Пример надуманный и примитивный, но что-то подобное может встретиться: внутри делаются большие вычисления, которые не хотелось бы повторять в тесте. И в данном примере хочется ввести строку покороче (чтобы число повторений в permutations, len(name), было меньше), но это запрещено. Можно было бы разбить функцию на две, вынести вызов permutations наружу, и в функцию передавать её вывод, но можно сделать по-другому.  
  
Вместо переписывания кода, мы можем просто «пропатчить» функцию permutations на время вызова, задать только определённый вывод и вызвать какой-то код:

from mock import patch

import itertools # важно: мы импортируем модуль, не сам метод

name = 'достаточно длинное имя'

>>> with patch('itertools.permutations') as perm\_mock:

... perm\_mock.return\_value = xrange(3)

... real(name)

1

3

6

Заметьте: print вызван вместо [*42! / (42 — 10)!*](http://docs.python.org/library/itertools.html#itertools.permutations) раз всего 3, то есть цикл пробежался по xrange(3), который мы подставили.  
  
Кроме того, после выхода из контекста **with** функция itertools.permutations вернулась в своё нормальное состояние.

Отладочная информация

Допустим, нужно проверить, что происходит с объектом, который мы передали функции. В той ли последовательности, с теми ли параметрами вызываются методы, к тем ли атрибутам обращаются. Для нужно просто запустить в неё объект Mock, который запишет всё, что происходит.  
  
Похожий пример из жизни: когда шли аэродинамические испытания Бурана, весь корабль не продували в трубе (таких не бывает) и не запускали в атмосферу. В воздухе летал специальный прототип БТС-002, а для отработок посадки использовался переоборудованный Ту-154 в обвесе, повторяющем аэродинамику Бурана.  
  
  
  
У объекта Mock есть несколько атрибутов с информацией о вызовах:

* called — вызывался ли объект вообще
* call\_count — количество вызовов
* call\_args — аргументы последнего вызова
* call\_args\_list — список всех аргументов
* method\_calls — аргументы обращений к вложенным методам и атрибутам (о них — ниже)
* mock\_calls — то же самое, но вместе и для самого объекта, и для вложенных

В нашем примере выше можно убедиться, что функция real() правильно вызывает permutations. Чтобы ещё точнее проверить, например, в автоматических тестах, можно вызвать один из [методов assert\_\*](http://www.voidspace.org.uk/python/mock/mock.html#mock.Mock.assert_called_with):

perm\_mock.assert\_called\_with(xrange(len(name)), 10)

Синтаксический сахар

Для юнит-тестов в Джанго пригодится то, что patch работает как декоратор:

@patch('itertools.permutations')

def test(ip):

ip.return\_value = range(5)

print list(itertools.permutations(xrange(10), 10))

>>> test()

[0, 1, 2, 3, 4]

Макеты атрибутов и цепочек вызова

Иногда в Джанго нужно делать что-то с файлами, и лучше обойтись без сохранения их на диск или другое хранилище. Обычным путём мы бы наследовали от класса File и перезаписали бы некоторые свойства, что было бы громоздко. Но в Mock можно описать сразу и атрибуты, и цепочки вызова:

mock\_file = Mock()

mock\_file.name = 'my\_filename.doc'

mock\_file.\_\_iter\_\_.return\_value = ['строка 1', 'строка 2', 'строка 3']

stat = mock\_file.stat.return\_value

stat.size, stat.access\_time = 1000, datetime(2012, 1, 1)

Вот сколько тестового кода было сэкономлено. Кстати, передавая объект как аргумент или ожидая объект из функции, полезно дать им имена, чтобы отличать:

>>> mock\_a = Mock(name='макет файла')

>>> mock\_a

<Mock name='макет файла' id='169542476'>

Как же эти цепочки атрибутов работают?

>>> m = Mock()

>>> m

<Mock id='167387660'>

>>> m.any\_attribute

<Mock name='mock.any\_attribute' id='167387436'>

>>> m.any\_attribute

<Mock name='mock.any\_attribute' id='167387436'>

>>> m.another\_attribute

<Mock name='mock.another\_attribute' id='167185324'>

Как видите, обращение к атрибуту выдаёт ещё один экземпляр класса Mock, а повторное обращение к тому же атрибуту — снова тот же экземпляр. Атрибут может быть чем угодно, в том числе и функцией. Наконец, любой макет можно вызвать (скажем, вместо класса):

>>> m()

<Mock name='mock()' id='167186284'>

>>> m() is m

False

Это будет другой экземпляр, но если вызвать ещё раз, экземпляр будет тем же самым. Так мы можем назначить этим объектам некоторые свойства, после чего передать этот объект в тестируемый код, и они там будут считаны.  
  
Если мы назначим атрибуту значение, то никаких сюрпризов: при следующем обращении получим именно это значение:

>>> m.any\_attribute

<Mock name='mock.any\_attribute' id='167387436'>

>>> m.any\_attribute = 5

>>> m.any\_attribute

5

Полезно прочесть в [документации класса](http://www.voidspace.org.uk/python/mock/mock.html), какие атрибуты у него родные, чтобы не случалось коллизии названий.

Как ограничить гибкость макета

Как видите, можно обращаться к любому атрибуту макета, не вызывая ошибки AttributeError. У этого удобства есть обратная сторона: что если мы поменяем API, например, переименуем метод, а функция, работающая с нашим классом, будет обращаться к прежнему методу? Код на самом деле не работает, а тест выполняется без ошибок. Для этого можно задать спецификацию объекта в параметре **spec** (либо классу Mock, либо patch), и макет будет вызывать ошибку при обращении к несуществующим свойствам:

class Tanchik(object):

model = 'T80'

def shoot(self, target):

print 'Бдыщь!'

def tank\_sugar(target):

print '%s стреляет' % tank.model

tank.shoot(target)

return tank

==================

import tanks

@patch('tanks.Tanchik', spec=tanks.Tanchik) # <<== задано свойство spec

def test\_tank(tank\_mock):

assert isinstance(tank\_sugar(tank\_mock), tanks.Tanchik)

Теперь если мы переименуем model или shoot у танчика, но забудем исправить tank\_sugar, тест не выполнится.

Как сделать макет умнее

Хорошо, допустим, Mock умеет заменять нужные объекты на ненужные и подменять вывод. А можно ли подменить функцию на что-то более сложное, чем значение (return\_value)? Есть 2 пути:

* если нужно переопределить много методов у экземпляра или класса, наследуем от класса Mock
* если нужно заменить только один вызов (метод или сам класс), используем **side\_effect**.

def simple\_function(args):

do\_something\_useful()

with patch('module.BigHeavyClass', side\_effect=simple\_function) as mock\_class:

another\_class.take(mock\_class)

Кстати, не нужно писать в simple\_function контрольный вывод, потому что, как сказано выше, в конце кода в объекте mock\_class можно считать method\_calls.

Подмена встроенных функций

Сам по себе Mock не может заменить встроенные в язык функции (len, iter), но может сделать [макет с нужными «волшебными» функциями](http://www.voidspace.org.uk/python/mock/magicmock.html#mocking-magic-methods). Например, вот мы делаем макет файла:

>>> mock = Mock()

>>> mock\_bz2module.open.return\_value.\_\_enter\_\_ = Mock(return\_value='foo')

>>> mock\_bz2module.open.return\_value.\_\_exit\_\_ = Mock(return\_value=False)

>>> with mock\_bz2module.open('test.bz2') as m:

... assert m == 'foo'

Для множества подобных случаев, когда требуется эмулировать стандартный объект (список, файл, число), есть класс **[MagicMock](http://www.voidspace.org.uk/python/mock/magicmock.html" \l "magic-mock)** с набором значений, пригодных для тестов.

Где Mock не работает

Сам автор модуля, Майкл Фурд, говорит, что принцип, где нужны макеты, а где нет, простой: если с макетами тестировать проще, их надо использовать, а если с ними труднее, надо отказаться.  
  
Бывает так, что нужно тестировать связку двух модулей, нового и старого, и в связке много перекрёстных вызовов. Нужно внимательно смотреть: если мы постепенно начинаем переписывать поведение целого модуля, пора остановиться — код теста жёстко сввязан с кодом модуля, и при каждом изменении в рабочем коде придётся менять и тесты. Кроме того не стоит пытаться написать целый модуль-макет вместо старого.  
  
По моему личному опыту, Mock может конфликтовать с отладчиками, например, в [PuDB](http://pypi.python.org/pypi/pudb) случалась бесконечная рекурсия. [IPDB](http://pypi.python.org/pypi/ipdb) работал нормально, поэтому тесты проекта мы выполняли с IPDB, а просто код — на PuDB.

Выводы

Макеты в Mock можно подставлять всюду и как угодно. Ваш код не придётся подстраивать под тестирование, что значит быстрее разработка, и, возможно, быстрее прототипирование.  
  
Можно выбросить из тестов всё лишнее, всё занимающее время и ресурсы, оставив работать только тот код, который нужно проверить.  
  
Настройки макетов где нужно жёсткие (spec), где нужно гибкие, и патчи не оставляют за собой следов.