13. (Още) Unit тестване

21 ноември 2022

Питанка

Какво ще изведе следният Python код:

```
pattern = Pattern.compile("[0-9]+", Pattern.CASE_INSENSITIVE);
matcher = pattern.matcher("abc00123xyz456_0");
match = matcher.matches();
```

А така?

```
System.out.println("matches() found nothing");
```

Това е Java, обиждате Жорката...

Питанка \d

Какво ще изведе следният (наистина) Python код:

```
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence)
print(matched.groups())
```

```
('we', 'are', 'humans')
```

Питанка \d\.\d

Какво ще изведе следният (отново наистина?) Python код:

```
matches = re.finditer('\d+', '12 drummers drumming, 11 ... 10 ...')
print(matches)
```

```
<callable_iterator object at 0x00000163C89CF1F0>
```

А този?

```
for match in matches:
    print(match.span())
```

```
(0, 2)
(22, 24)
(29, 31)
```

(Още) Unit testing

Да си припомним:

1. `import unittest`
2. `unittest.TestCase`
3. `setUp / tearDown`
4. `assert*`
5. Всеки тест **трябва** да започва с `test_`.

(Още) Unit testing

(Още) Да си припомним:

- Пишете тестове за всичко, което може да се счупи.
- Не пишете тестове, които **винаги** ще минават, дори и да счупите целия codebase.
- Добър начин да си мислите за тестовете е като requirements.
- *Не тествайте елементарен код (в italic, защото е субективно и защото ще го нарушим... малко)*.
- Успехът на тестовете не трябва да зависи от реда им.
- Тествайте гранични случаи!
- Не правете тестовете зависими един от друг.

Дайте да понапишем няколко теста

- `Entity.take_damage()`
- `Entity.move()`
- `Entity.attack()`

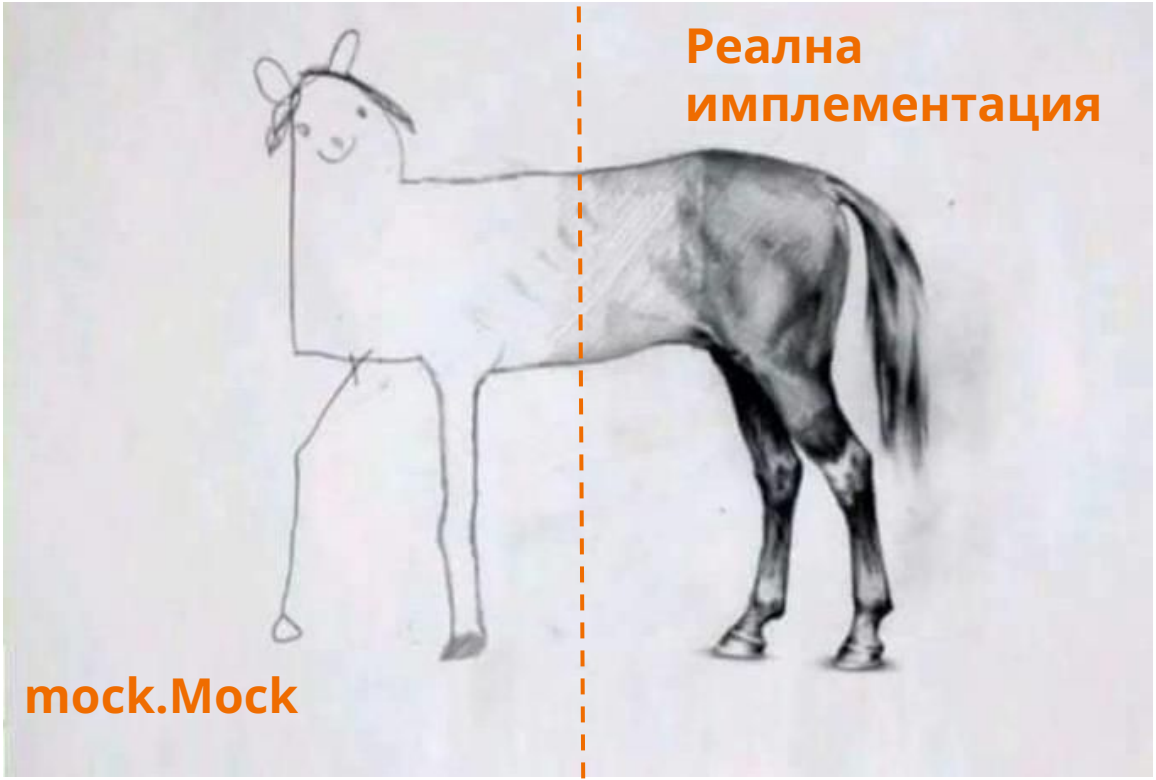
coverage.py

- ~~Инструмент, който измерва доколко е “изтестван” кодът ни...~~
- Не баш...
- Но пък измерва колко линии от кодът ни са били изпълнени по време на тестовете.
- С други думи, **не е гаранция**, че кодът ви е изтестван добре!
- `pip install coverage`
- `coverage run -m unittest discover`
- `coverage html`
- *(в нашия случай, с дървено сетъпната среда - `python -m coverage ...`)*
- Има и още други удобства, за които можете да прочетете по-долу:
  - https://coverage.readthedocs.io/en/7.3.2/

random.randint

- Опа, ами сега?
- Какво правим с нещо, което ни връща различен резултат всеки път?
- Като цяло - какво правим със зависимост към външен интерфейс (библиотека)?

unittest.mock



Enter unittest.mock

- Позволява ни да дефинираме “dummy” поведение, което да замести даден call.
- Позволява ни да raise-ваме exception при определени условия.
- Позволява ни да проверяваме дали и колко пъти са извикани методите, които mock-ваме.
- С други думи - позволява предвидимост на външните зависимости, които кодът ни има.
- Например?
- random.randint винаги да връща точно определено число!

Как да се подиграваме на кода си?

- `from unittest import mock`
- `mock.Mock`
- `mock.patch(target, ...)`
  - или
- `mock.patch.object(target, attribute, ...)`
- `assert_*`

Mock

```
from unittest.mock import Mock # В общия случай импортираме така
my_cool_mock = Mock()
print(my_cool_mock) # <Mock id='1528078724240'>
```

Какво можем да правим с него?

- Да го подаваме като аргумент.
- Да заместваме части от кода - външните зависимости.
- И то много гъвкаво.

Mock (2)

```
import random
random.randint(1, 10) # 3
mock_randint = Mock()
random.randint = mock_randint
random.randint # <Mock id='1706764146960'>
random.randint(1, 10) # <Mock name='mock()' id='1706764147968'>
mock_randint.assert_called_once() # Нищо, защото е вярно
mock_randint.assert_called_once_with((1, 15))
```

```
Traceback (most recent call last):
  ...
AssertionError: expected call not found.
Expected: mock((1, 15))
Actual: mock(1, 10)
```

Имаме и още удобства

```
... # Същия сетъп
random.randint(0, 10)
random.randint(10, 50)
mock_randint.call_count # 2
mock_randint.call_args # call(10, 50) - Последното извикване
mock_randint.call_args_list # [call(1, 10), call(10, 50)] - Всички извиквания


print(dir(mock_randint)) # И още бая неща

['assert_any_call', 'assert_called', 'assert_called_once',
'assert_called_once_with', 'assert_called_with', 'assert_has_calls',
'assert_not_called', ...]
```

Ретърн Вальо

Супер, обаче на нас ни трябва да го модифицираме, така че да има определено поведение.

```
import random
random.randint(1, 10) # 3
mock_randint = Mock()
mock_randint.return_value = 'Вальо'
random.randint = mock_randint
random.randint(1, 10) # 'Вальо'
random.randint(1, 10) # 'Вальо'
random.randint(1, 5) # 'Вальо'
random.randint('piet i siediem') # 'Вальо'
```

- Забележете, че аргументите, с които извикваме даже нямат значение.
- `Mock` е толкова гъвкав, че понякога може да се окаже прекалено гъвкав!
- Горният “проблем” си има решение, но за тази лекция няма да ни вълнува.

P.S. Всъщност можем да mock-нем цяла библиотека

```
import xml
mock_xml = Mock()
mock_xml.return_value = "XML-a ti e top, bace, evalata"
xml = mock_xml
print(xml.etree.ElementTree.parse({'not_xml': 'actually_json'})) # 'XML-a ti e
top, bace, evalata'
print(xml.etree) # <Mock name='mock.etree' id='1706764151136'>
print(xml.etree.ElementTree) # <Mock name='mock.etree.ElementTree'
id='1706764151088'>
print(xml.does_not_exist.com) # <Mock name='mock.does_not_exist.com'
id='1706763122496'>
```

- С други думи, всяко име, което потърсим, също ще бъде mock-нато.
- Може да ви изглежда sketchy, но всъщност е доста важно поведение.

Последно за Mock

```
import xml.etree.ElementTree as ET
mock_xml_parse = Mock()
mock_xml_parse.side_effect = TypeError
ET.parse = mock_xml_parse
ET.parse('actually_valid_xml.xml')
```

```
    Traceback (most recent call last):
    ...
    TypeError
```

С други думи - `side_effect` може да се използва за да се извика дадена функция или по-често - да хвърли някакъв `Exception`.

Излъгах, но това е последно, наистина

```
import random
mock_randint = Mock(return_value='Вальо')
random.randint = mock_randint
```

```
import random
mock_gauss = Mock(side_effect=ZeroDivisionError)
random.gauss = mock_gauss
```

- По-лесен начин, да си конфигурираме Mock-а.
- *class unittest.mock.Mock(spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs)*

Проблем

Това:

```
random.randint = mock_randint
```

Не е яко!

На практика затриваме истинския `random.randint`.

patch

Все пак имаме начин да заместим зависимостта само временно!

```
import random
from unittest.mock import patch, Mock
mock_randint = Mock(return_value=42)
print(random.randint) # <bound method Random.randint of <random.Random object
at 0x00000163C640CAD0>>

with patch('random.randint', mock_randint):
    print(random.randint)    # <Mock id='1528075649104'>
    print(random.randint())  # 42

print(random.randint) # <bound method Random.randint of <random.Random object
at 0x00000163C640CAD0>>
```

patch (2)

- `patch` ни позволява да заместим необходимите интерфейси в даденият namespace *(заместваме “randint” в namespacea “random” - да, модулите са на практика namespaceове)* само в блока код непосредствено след употребата му.
- Може да се ползва като context manager (което вече видяхме).
- Може да се ползва и като декоратор:

```
@patch('random.randint', mock_randint)
def very_random():
    print(random.randint)
    print(random.randint())

very_random()
# <Mock id='1528075649104'>
# 42
```

patch.object

Същото, но му подаваме **обект** и **име**, което да mock-не.

```
with patch.object(random, 'randint', mock_randint):
    print(random.randint)    # <Mock id='1528075649104'>
    print(random.randint())  # 42
```

Мързеливо mock-ване

- Според примерите до момента имаме нужда от експлицитно дефиниране на `Mock` обект...
- Е, има и по-лесен начин:

```
with patch('random.randint', return_value=42):
    print(random.randint)    # <MagicMock name='randint' id='1528077220560'>
    print(random.randint())  # 42
```

- О не, какво е `MagicMock`?!
- Спокойно, същото като `Mock`, просто автоматично имплементира всички дъндъри (или другояче казано - магически методи).
- В общия случай е по-полезното от двете, но предвид горния пример - по-рядко ще ви се налага да го дефинирате експлицитно.

Добре, но как достъпваме обекта?

Ако искаме да проверим `call_args`, да правим `assert_*` проверки и прочие, просто добавяме, стандартно за context manager-ите - `as xx`.

```
with patch('random.randint', return_value=42) as mock_randint:
    x = random.randint(1, 50)

print(x) # 42
mock_randint.call_count # 1
mock_randint.call_args_list # [call(1, 50)]
```

- Същото можете да правите със `side_effect` и каквото още се сетите.
- `unittest.mock.patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

Добре, вече можем да се върнем на Entity.attack()

Вие диктувате, аз пиша...

Проблеми при mock/patch-ване

- Важно е да знаем кога точно да mock/patch-нем. Това рядко ще го бъркате.
- Важно е да знам какво/къде точно да mock/patch-нем. Това често ще го бъркате - вече илюстрирахме, че е пипкаво да заместим в правилния namespace, а нашият пример дори не беше кой знае колко сложен. При по-сложни зависимости и импорти е възможно да се окаже още по-неприятно, но с малко гледане на импорти и имена ще се оправите.
- Важно е да знаем докъде да си ограничим mock/patch-ването. Това средно често ще го бъркате - представете си, че имаме зависимост към функция от `math`, и решим да заместим целия модул... И после се окаже, че има нещо от модула, което всъщност ни е важно и не искаме да го mock/patch-ваме. Have fun debugging. :p

Проблеми при mock/patch-ване (2)

- Ако променяте имената на методите - не забравяйте колко гъвкав е `Mock`. Има шанс някакви неща да продължат да са верни дори и след това - например `assert_not_called`, когато решите да смените името на метода... Ще продължи да е вярно.
- В този ред на мисли - `my_mock.asert_called()` никога няма да ви фейлне. Защо?
- За горното, прочетете за атрибута на `Mock` и `patch` - `spec`.
- И като цяло:
  - https://docs.python.org/3/library/unittest.mock.html

А кодът (тестовете) от лекцията

- Тук:
  - https://github.com/vbechev/ascii-poop

Въпроси?