11. Автоматизирано тестване

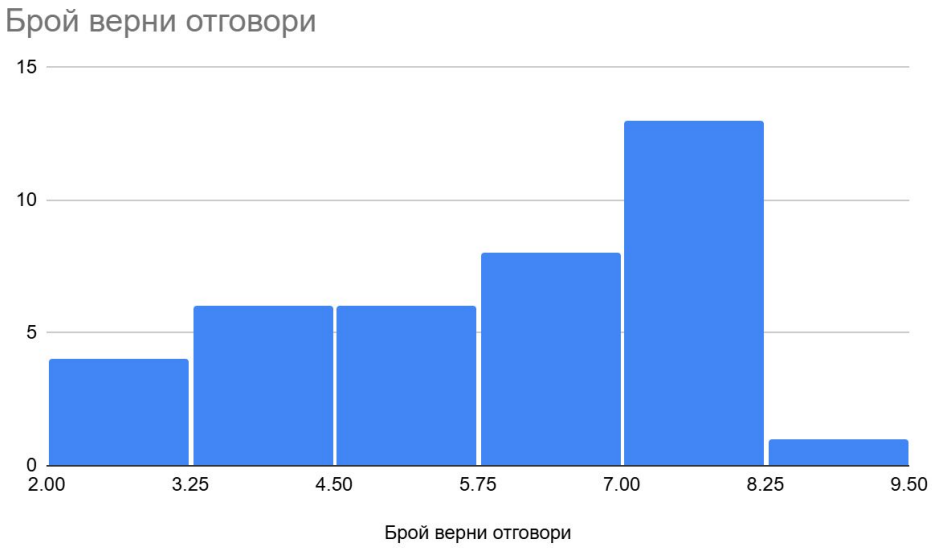
1 април 2026

Малко информация от контролното!

- Вероятно вече сте видели, че имате качени в сайта точки
- Както сме казали, скалираме точките спрямо най-добрия резултат
- Т.е. не е непременно да имате 30 верни отговора за да имате 30 точки
- Което се оказва проблем...

:\

- Най-високият брой верни отговори на контролното е **9/30**
- И то само от един човек
- С други думи със 6 верни отговора получавате 20 точки
- Което не изглежда особено добре
- Но it is what it is
- И да, проверихме два пъти, и само един от двата пъти проверяващият беше пиян
- Другата лекция ще обсъдим най-честите грешки
- Да продължаваме нататък

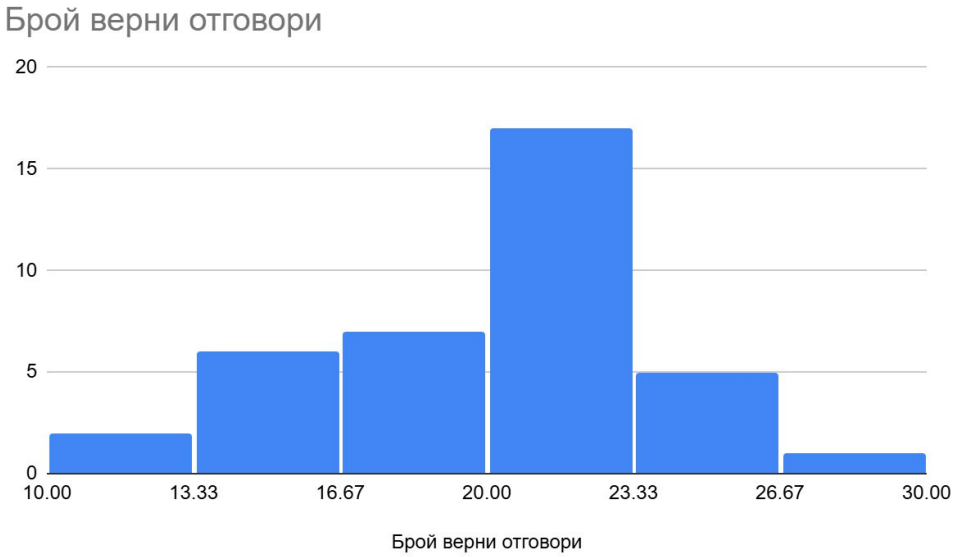


матизирано тестване

1 април 2026

Сега сериозно

- Всъщност сте се представили доста добре
- Медианата е **20** верни отговора
- Максимумът е **27/30** *(игнорирайте странно изглеждащата графика, може и по-добре)*
- Браво!
- И следващия път наистина ще си поговорим за най-масово сгрешените въпроси
- ☺️



А сега, три въпроса

Какво може да бъде модул в python?

- Файл с разширение `.py`
- Директория съдържаща файл с име `__init__.py`

Но първо, три въпроса - 2

```
# pythons.py
pythons_by_area = {'Arnhem': 'Oenpelli', 'Bismark Islands': 'Bothrochilus',
'New Guinea': 'Apodora'}

# python_classifier.py
import pythons
def python_in_area(area):
    return pythons.pythons_by_area[area]

# hmmmm.py
import pythons
import python_classifier
pythons.pythons_by_area['Arnhem'] = 'Nyctophilopython'

print(python_classifier.python_in_area('Arnhem')) # ??
'Nyctophilopython'
```

Още по темата (цък)

Но първо, три въпроса - 3

```python
# legit_business.py
try:
    from colombia import *
except ImportError:
    from colombia import cocaine as
keistered_goods
    cocaine = keistered_goods

print(cocaine)
# ??
```

```python
# colombia.py
__all__ = ["cocaine", "undercover_cops"]
undercover_cops =
    __import__("the_po_po").undercover
cocaine = "The good stuff!"
```

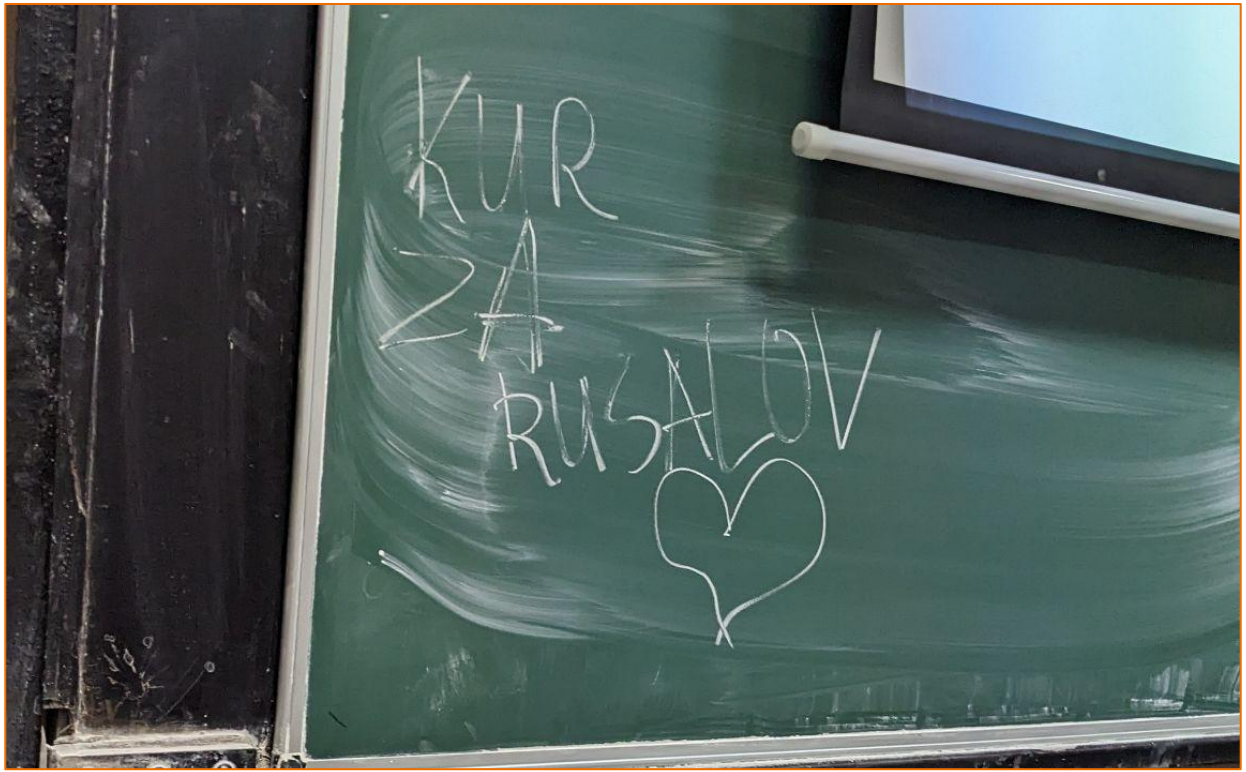
```
Traceback (most recent call last):
  File "d:\Dev\FMI\Python\modules_example\legit_business.py", line 4, in <module>
    from colombia import cocaine as keistered_goods
  File "d:\Dev\FMI\Python\modules_example\colombia.py", line 2, in <module>
    undercover_cops = __import__("the_po_po").undercover
                      ^^^^^^^^^^^^^^^^^^^^^^
ModuleNotFoundError: No module named 'the_po_po'
```

Но първо, три въпроса - 4 (хехе)

```
numbers = range(10, 20)
for number in numbers:
    if number > 20:
        print("It's too big!")
        break
else:
    print("It's not too big.")

# ??

"It's not too big."
```

Просто искахме да ви споменем, че `for` има `else`... Не че сме го ползвали повече от два пъти...

Тро-ло-ло: Георги тролва Виктор (и Русалов)



Не, това не е мое становище. Вие го пишете по дъските!

За какво няма да си говорим

- За quality assurance
- За acceptance testing
- За тестове тип “бенчмарк”
- За сомелиерство на ракия

Митът

- Проектът идва с готово, подробно задание
- Прави се дизайн
- С него работата се разбива на малки задачи
- Те се извършват последователно
- За всяка от тях пишете кода
- Разцъквам го малко - няколко print-а, малко пробване в main метода/функцията и толкова
- Profit

Реалността

- Това в началото с ясните изисквания и дизайн е утопия
- Писането на код е сложна задача - допускат се грешки
- Програмистите са хора - допускат грешки
- Промяната на модул в единия край на системата като нищо може да счупи модул в другия край на системата
- Идва по-добра идея за реализация на кода
- Често се налага един код да се преработва

Как да автоматизираме

- За всичко съмнително ще пишем **сценарий**, който да "цъка"
- Всеки сценарий ще изпълнява кода и ще прави няколко **твърдения** за резултатите
- Сценариите ще бъдат обединени в **групи**
- Пускате всички тестове с едно бутонче
- Резултатът е "Всичко мина успешно" или "Твърдения X, Y и Z в сценарии A, B и C се оказаха неверни"

Например

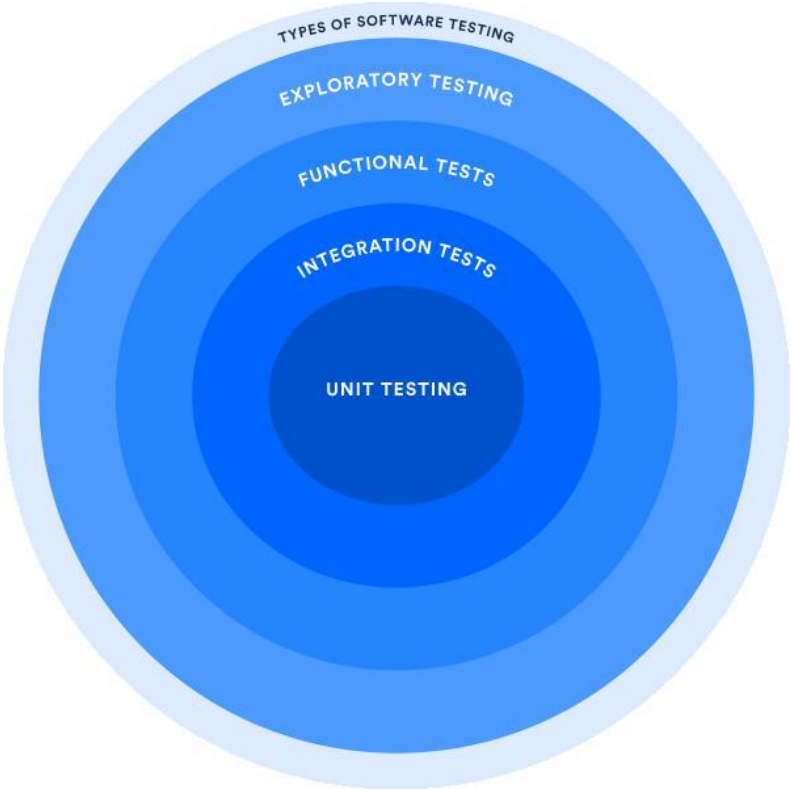
```
.F....F.....
======================================================================
FAIL: test_mixed_sentence (test.TestSentence)
Test with mixed remainder input.
----------------------------------------------------------------------
Traceback (most recent call last):
File "/tmp/test.py", line 75, in test_mixed_sentence
self.assertEqual(split_sentence('Здравейте момчета къде ми е отвертката'),
AssertionError: Lists differ: [('Зд[49 chars]', 'е'), ('м', '', 'и'), ('', '', 'е'), ('отв', 'ертк', 'ата')] != [('Зд[49 chars]', 'е'), ('м', '', 'и'), ('', 'е', ''), ('отв', 'ертк', 'ата')]

First differing element 4:
('', '', 'е')
('', 'е', '')
```

За какво ни помагат тестовете

- Откриват грешки по-рано
- Позволяват ни уверено да правим промени в системата
- Дават сигурност на клиенти, шефове и програмисти
- Представляват пример как се работи с кода
- Служат като документация и спецификация
- Дават гаранции, които в не-динамични езици получаваме при компилация

Типове тестове

- При преминаване към по-външните кръгове на *(ада)* тестването започваме да работим в условия, по-близки до реалните
- **Unit** тестовете трябва да се концентрират върху изолирана *(малка)* част от функционалността и да верифицират правилната ѝ работа



Имаме 4 фази на тестване

- **Setup** - конфигурираме “контекста” на тестването
- **Execute** - реалното изпълнение на тестваното
- **Verify** - оценяваме дали крайният резултат е очакваният
- **Teardown** - връщаме “контекста”, променен по време на “Setup” фазата в начално състояние

По-конкретно?

1. `import unittest`
2. `unittest.TestCase`
3. `setUp / tearDown`
4. `assert*`
5. Всеки тест трябва да е напълно независим от останалите
6. `unittest.main()` = магия

```python
import unittest

class TestStringMethods(unittest.TestCase):

    def setUp(self):
        self.value = 'hmmmm'
        print(self.value)

    def tearDown(self):
        del self.value

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

if __name__ == '__main__':
    unittest.main()
```

Как се пишат тестове?

- `import unittest`
- `unittest.TestCase`
- `assertTrue(expression)`
- `assertFalse(expression)`
- `assertEqual(expected, actual)`
- `assertNotEqual(expected, actual)`
- `assertIs(expected, actual)`
- `assertIsNot(expected, actual)`
- `assertIsNone(expression)`
- `assertIsNotNone(expression)`
- `assertIn(element, collection)`
- `assertNotIn(element, collection)`
- `assertIsInstance(object, type)`
- `assertNotIsInstance(object, type)`
- И много други...

Само unittest?

- Добре де, има и друг вариант...
- `import pytest`
- Има разлики между двете
- Няма да задълбаваме в разликите, и двете са мощни и двете се използват масово

Кога да тестваме?

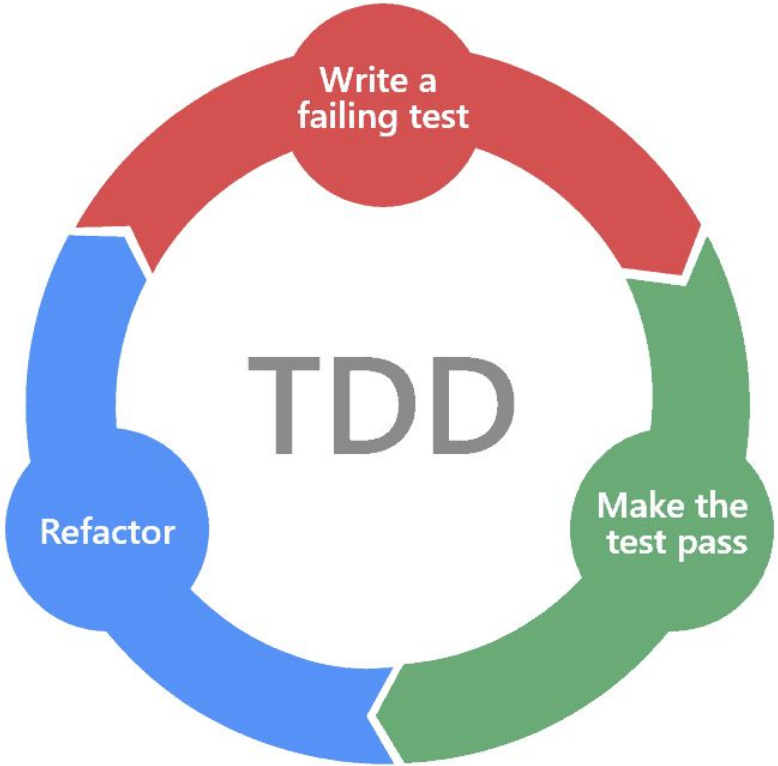
A. Никога
B. След като напишем функционалността
C. Преди да напишем функционалността
D. Когато ни е скучно и нямаме какво да правим

Според Test-Driven Development парадигмата, отговорът е:
C. Преди да напишем функционалността
В зависимост от това колко точно скучно ни е, може и D.

TDD

- Test-Driven Development is not about testing
- Подход за писане на код
- Дизайнът е базиран върху обратна връзка, не гадаене
- Спестява излишен код - пишете само каквото ви трябва
- Спестява излишна функционалност

TDD (наглядно)



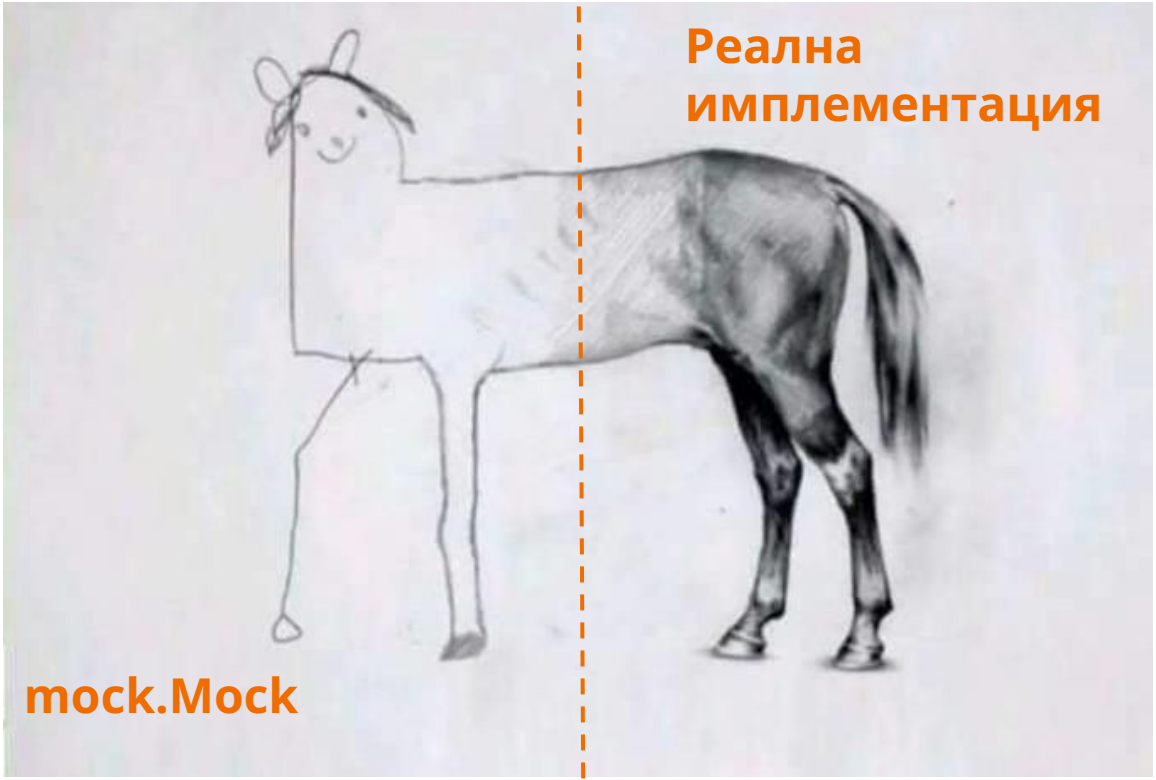
coverage.py

- ~~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 хвърляме грешка 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(0, 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('В кого е тръбата?') # '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()
xml = mock_xml
print(xml.etree.ElementTree.parse({'not_xml': 'actually_json'})) # <Mock
name='mock.etree.ElementTree.parse()' id='1633284230128'>
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)`

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

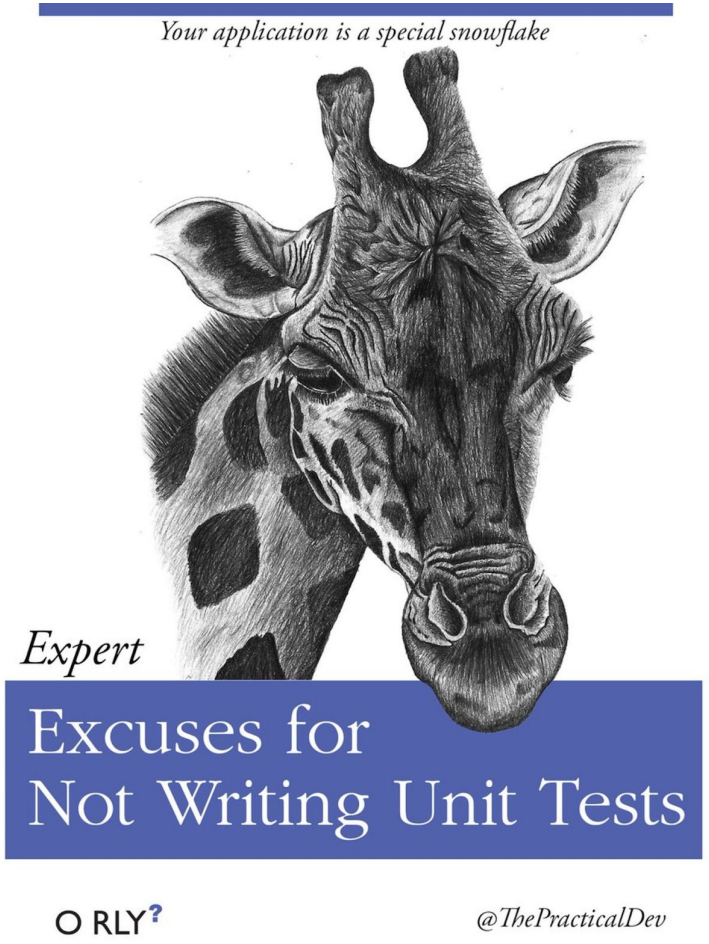
Проблеми при mock/patch-ване (2)

- 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

Добри практики при писане на тестове

- 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
- В този ред на мисли - не тествайте спрямо вътрешната имплементация на обектите
- Не тествайте елементарен код
- Не тествайте пили
- Успехът на тестовете не трябва да зависи от реда им
- Тествайте гранични случаи!
- Не правете тестовете зависими един от друг

Welcome to the real world...



Повторението е майка...

(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
- В този ред на мисли - не тествайте спрямо вътрешната имплементация на обектите
- *Не тествайте елементарен код (в italic, защото е субективно и ще го нарушим малко...)*
- Не тествайте пили
- Успехът на тестовете не трябва да зависи от реда им
- Тествайте гранични случаи!
- Не правете тестовете зависими един от друг

Въпроси?