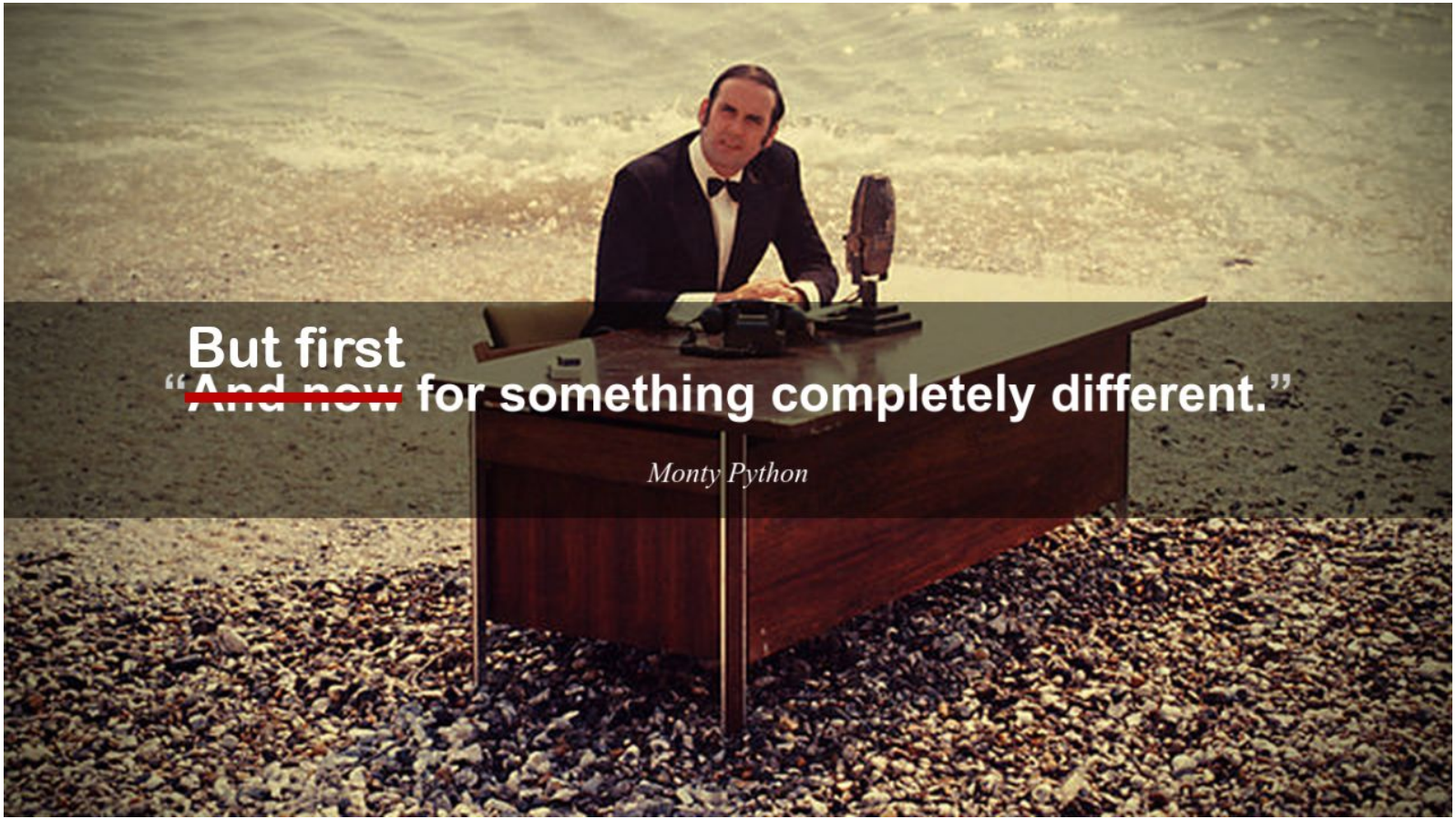
12. Автоматизирано тестване

19 ноември 2024


But first
"And now for something completely different."
Monty Python

11 ½. Да довършим от предния път

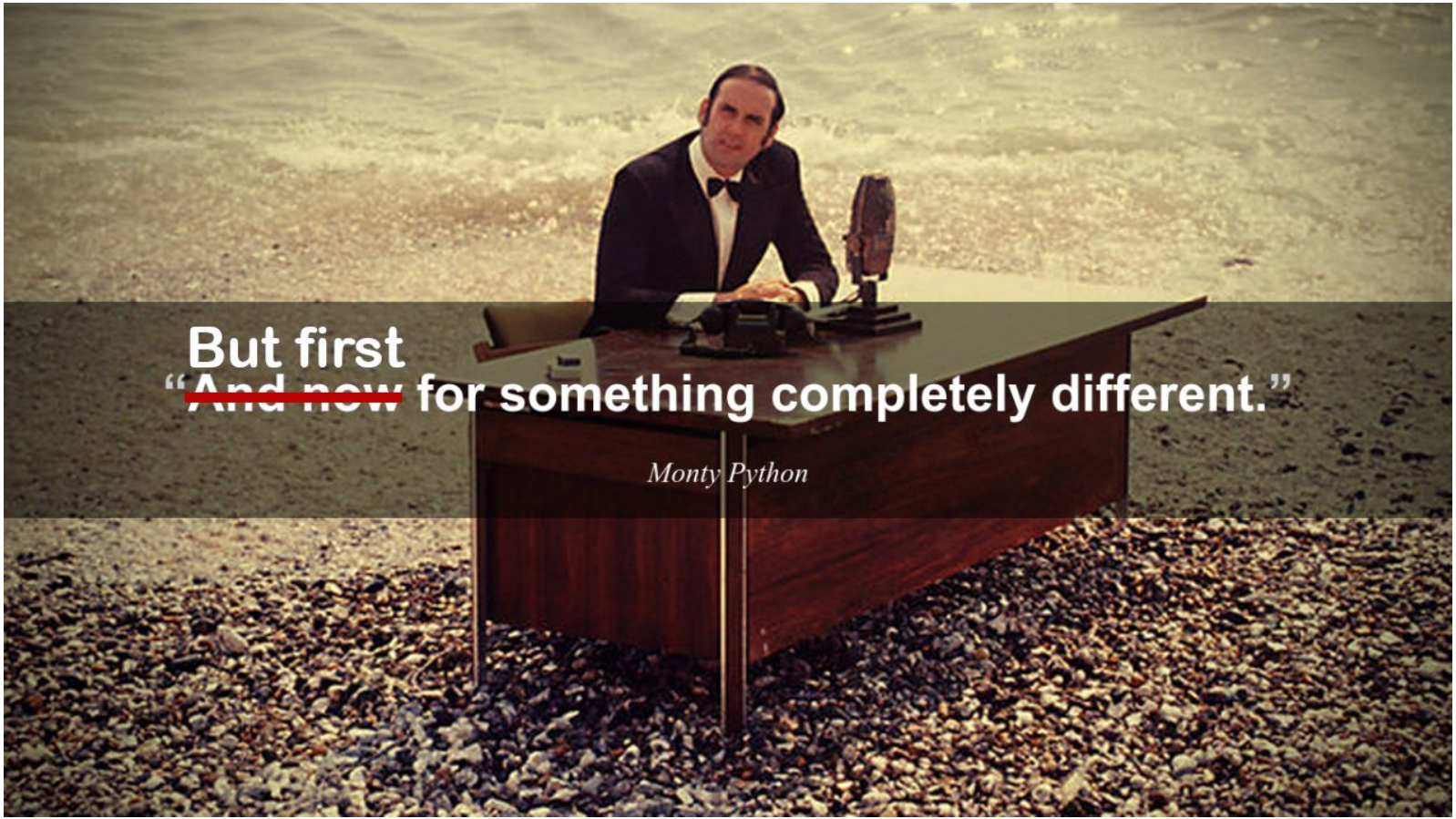
14? ноември 2024

Виртуална среда

- Ще направим едно ново репозитори в GitHub
- Ще го клонираме локално
- Ще създадем виртуална среда
- Ще инсталираме coolprint
- Ще напишем няколко реда код с него
- Ще направим requirements.txt
- Ще напишем README.md
- Ще качим всичко в GitHub

12. Автоматизирано тестване

19 ноември 2024


But first
"And now for something completely different."
Monty Python

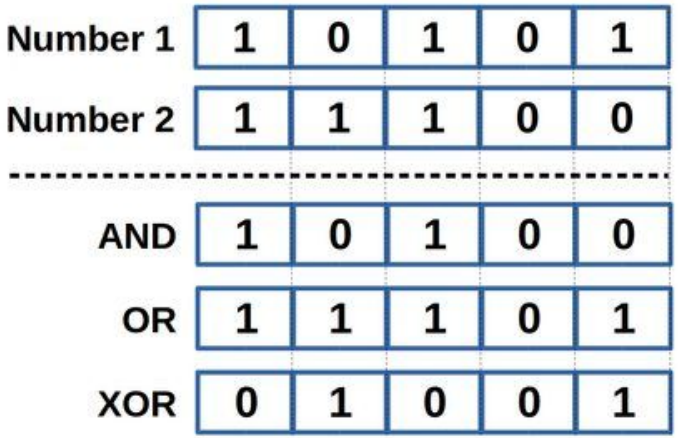
12?. ДеФЛОратори

a.k.a по-битови операции

защото всяка жена обича мъже, които
владеят “битовите операции”

Побитови операции

- Числата в програмирането се съхраняват като комбинация от 0 и 1-ци, това го знаете
- Обикновенно работим с human-readable числа, а не с нули и единици
- Какво правим ако искаме да пипнем по-дълбоко, обаче?
- Enter bitwise operations



&, |, ^, ~, >>, <<, (ノ °□°)ノ ⌒ ┻━┻

- Всяка от тези операции работи директно с двоичната репрезентация на числата
- & - побитово (логическо) **И**
- | - побитово **ИЛИ**
- ^ - побитово изключващо или (**XOR**)
- ~ - побитово отрицание (**НЬЕ**)
- >> - **shift надясно**
- << - **shift наляво**
- (ノ °□°)ノ ⌒ ┻┻ **-** побитово ‘бал съм му майката

Особености

```
>>> bin(-2)
'-0b10'
>>> bin(-3)
'-0b11'
>>> -2 & -3
-4
```

- Отрицателните числа в Python са имплементирани използвайки механизмът two's complement
- Реално `-2` не е `-0b10`, това е human-readable двоична репрезентация
- Всъщност е `1.....11111111111110`
- Няма да навлизаме в детайли как работят компютрите, защото става дълбоко, но предлагаме добро четиво по темата

Особености 2

```
>>> 0.1 & 0.2
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    0.1 & 0.2
TypeError: unsupported operand type(s) for &: 'float' and 'float'
```

- Числата с плаваща запетая са имплементирани с експонента и мантиса
- Ерго побитовите операции нямат смисъл в този контекст

Защо бихме ги използвали?

- Защото пишем на друг език преди 20 години
- Защото са секси и мацките / пичовете (много сме прогресивни) им се кефят
- Защото искаме да слагаме ей такива коментари в кода си:

```
i  = * ( long * ) &y;        // evil floating point bit level hacking
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
y  = * ( float * ) &i;
y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
```

- Флагове
- `chmod 644` някой?

12. Автоматизирано тестване

19 ноември 2024

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

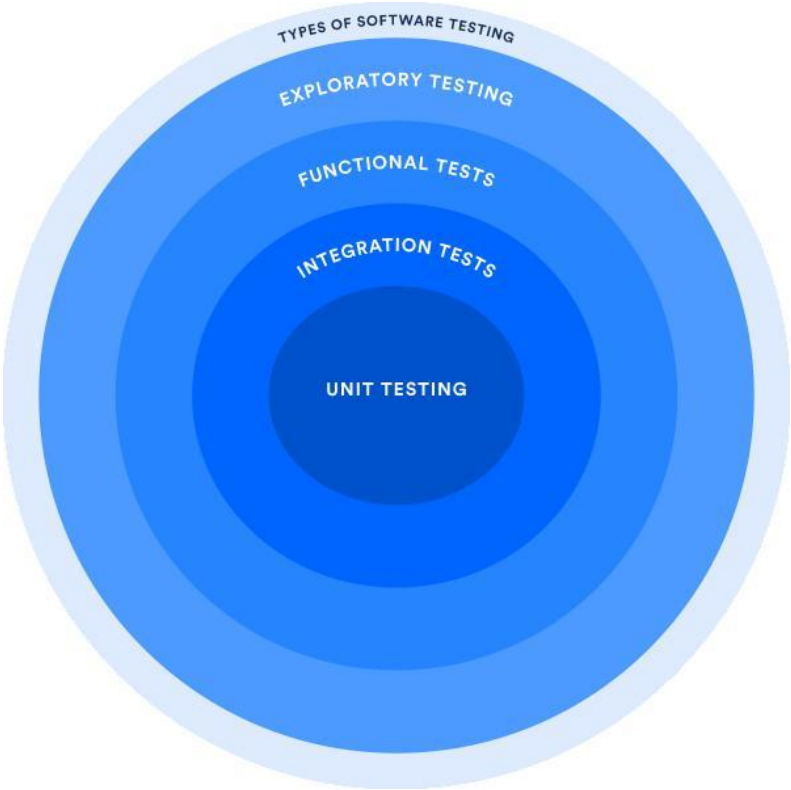
```
.F....F.....
======================================================================
FAIL: test_mixed_sentence (test.TestSentence)
Test with mixed remainder input.
----------------------------------------------------------------------
Traceback (most recent call last):
File "/tmp/test.py", line 75, 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

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?

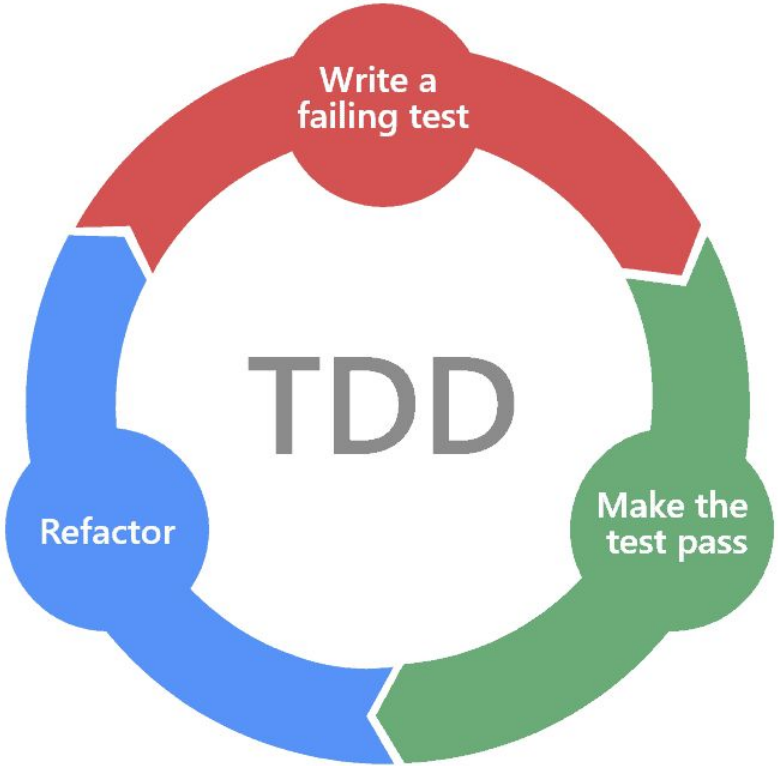
A. Никога
B. След като напишем функционалността
C. Преди да напишем функционалността
D. Когато ни е скучно и нямаме какво да правим

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

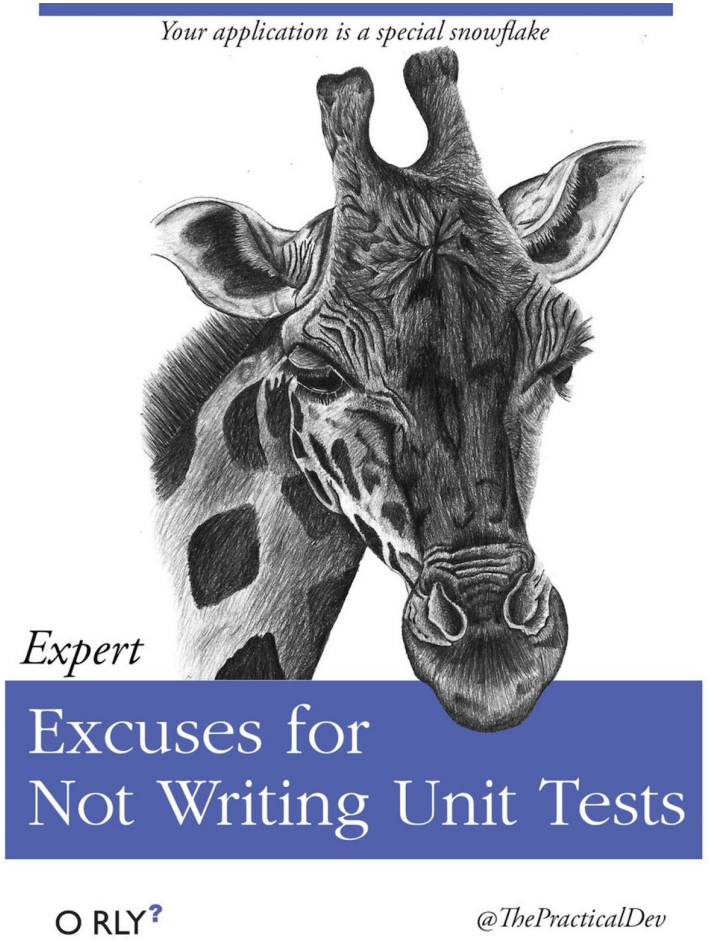
TDD (наглядно)



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...



Некст тайм - тестове по-нагледно нагледно

```python
class Character:
    _DAMAGE_MULTIPLIER = 5
    RESISTANCES = []

    def __init__(self, name, health, ac, fav_posish, level=1):
        self.name = name
        self.health = health
        self.__fav_posish = fav_posish
        self.level = level
        self.ac = ac + level
```

Въпроси?