PODSTAWY PROGRAMOWANIA W PYTHON

Dzień 12

AGENDA
DAY 12

- prywatność w Python
- testowanie kodu

pola i metody pseudo-prywatne

enkapsulacja

pola i metody pseudo-prywatne

Python daje możliwość stworzenia pseudo-prywatnych pól i metod.

Do nazwy (pola, metody) dodajemy dwa podkreślniki tylko z przodu.
Można je użyć wewnątrz klasy i w klasach dziedziczących, ale poza nią są niewidoczne – ale można i tak ich użyć!!!

```
self.__moje_pole_prywatne

def __metoda_prywatna(self, arg1):
    self.__moje_pole_prywatne = arg1
```

namespace

Namespace jest obszarem nazw, które są dostępne dla klasy.

```
print(MojaKlasa.__dict__)
print(instancja.__dict__)
```

W ten sposób możemy znaleźć pseudo-prywatny atrybut

properties: seter & getter

PROPERTIES

Properties – właściwości

definiujemy jak metody z dekoratorem, z nazwą identyczną jak zmienna, służą do manipulowania zmiennymi w kontrolowany przez nas sposób.

Wywołujemy bez nawiasów !!!!

PROPERTIES - GETTER

Getter – służy do zwrócenia wartości ze zmiennej

self.__name

```
@property
def name(self):
    return str(self.__name).capitalize()
```

PROPERTIES - SETTER

setter – służy do zapisania wartości do zmiennej – daje możliwość do kontrolowania tego co zapisujemy

self.__name

```
@name.setter
def name(self, name):
    self.__name = name
```

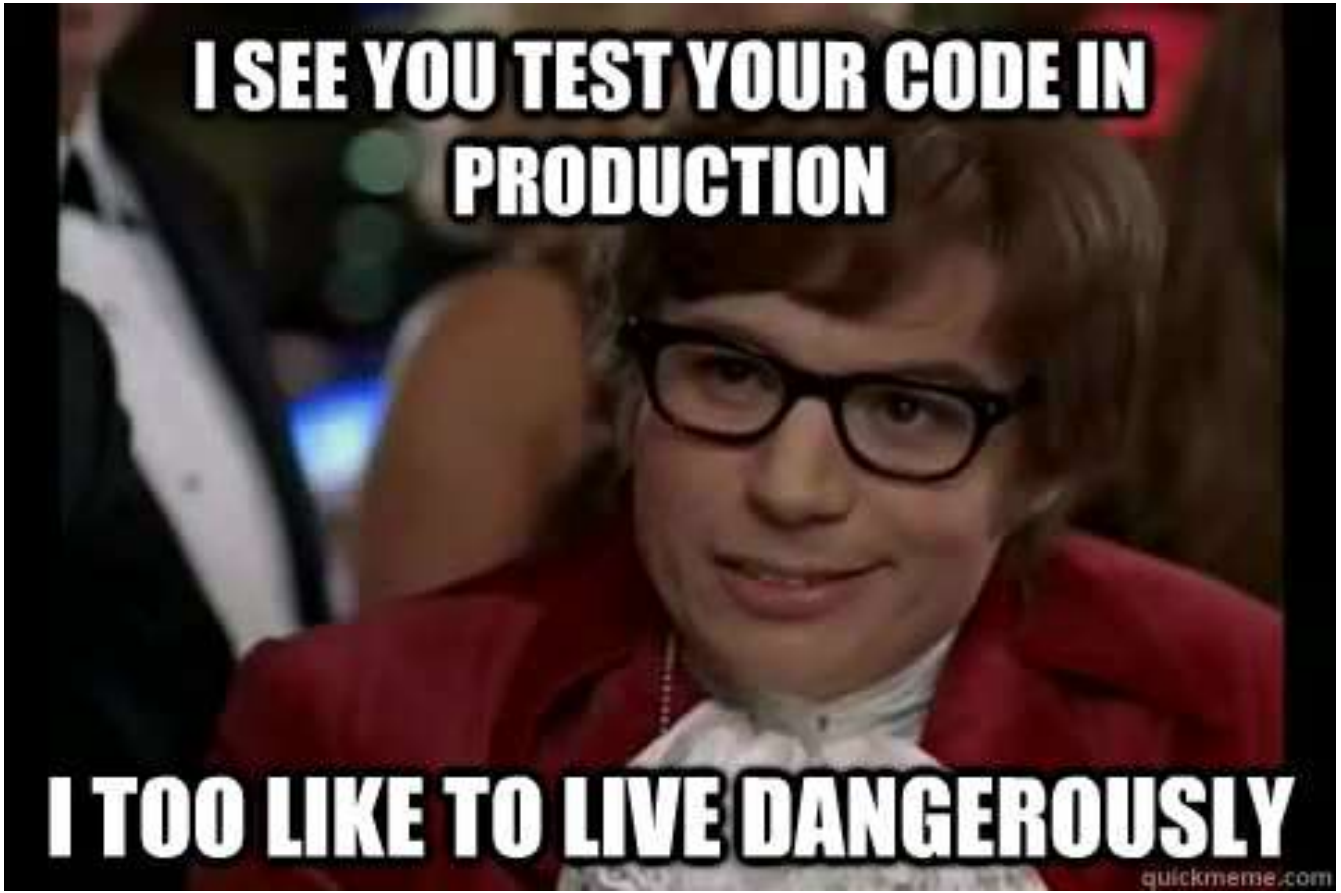
PROPERTIES - DELETER

setter – służy do usuwania zawartości zmiennej w kontrolowany sposób

self.__name

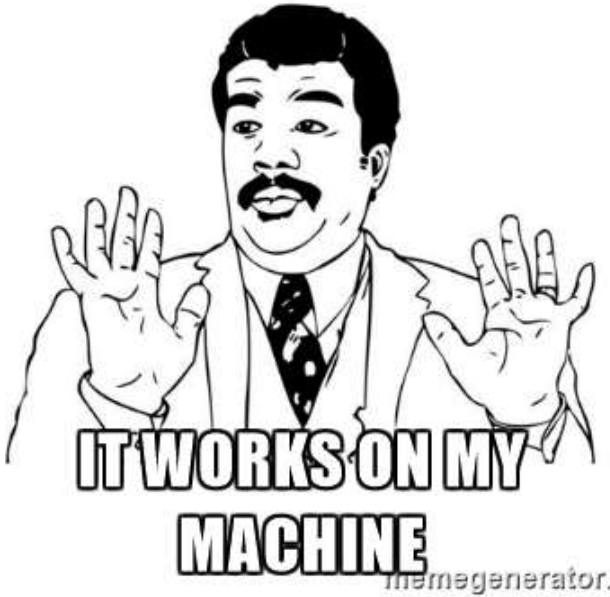
```
@name.deleter
def name(self):
    self.__name = None
```

testowanie kodu


I SEE YOU TEST YOUR CODE IN PRODUCTION
I TOO LIKE TO LIVE DANGEROUSLY
quickmeme.com

TESTY

Po co testować kod?



- sprawdzanie poprawności implementacji
- dbanie o stabilność i konsystentność kodu przy wprowadzaniu zmian,rozszerzaniu, refaktorowaniu
- odnajdywanie przypadków krańcowych
- niezależność od maszyny / środowiska developerskiego

TESTY

- **jednostkowe** – testujemy jednostkę logiczną naszego kodu
- integracyjne – sprawdzamy jak komponenty ze sobą współpracują
- systemowe – jak integracyjne, ale testujemy na poziomie systemów, usług zewnętrznych
- UI – testy interfejsu użytkownika

- manualne
- automatyczne

- black box
- white box

TESTY JEDNOSTKOWE

- testujemy poszczególne klasy, metody, properties

- muszą być niezależne od siebie
- muszą dawać ten sam rezultat niezależnie od kolejności wykonania

- możemy wspomóc się **code coverage** (pokrycie kodu testami) aby zobaczyć które części nie są przetestowane

- jeśli mamy sporo przypadków testowych dla jednego testu warto posłużyć się metodologią **DDT – Data Driven Tests** (testy sterowane danymi), najczęściej jako dekoratory metody testowej, lub zewnętrzny plik z przypadkami testowymi

TDD

Test Driven Development

Tworzenie kodu sterowane testami

- piszemy test
- uruchamiamy test
- test fail
- dopisujemy kawłek kodu
- uruchamiamy test…

- … test przechodzi, gdy cała funkcjonalność jest zaimplementowana

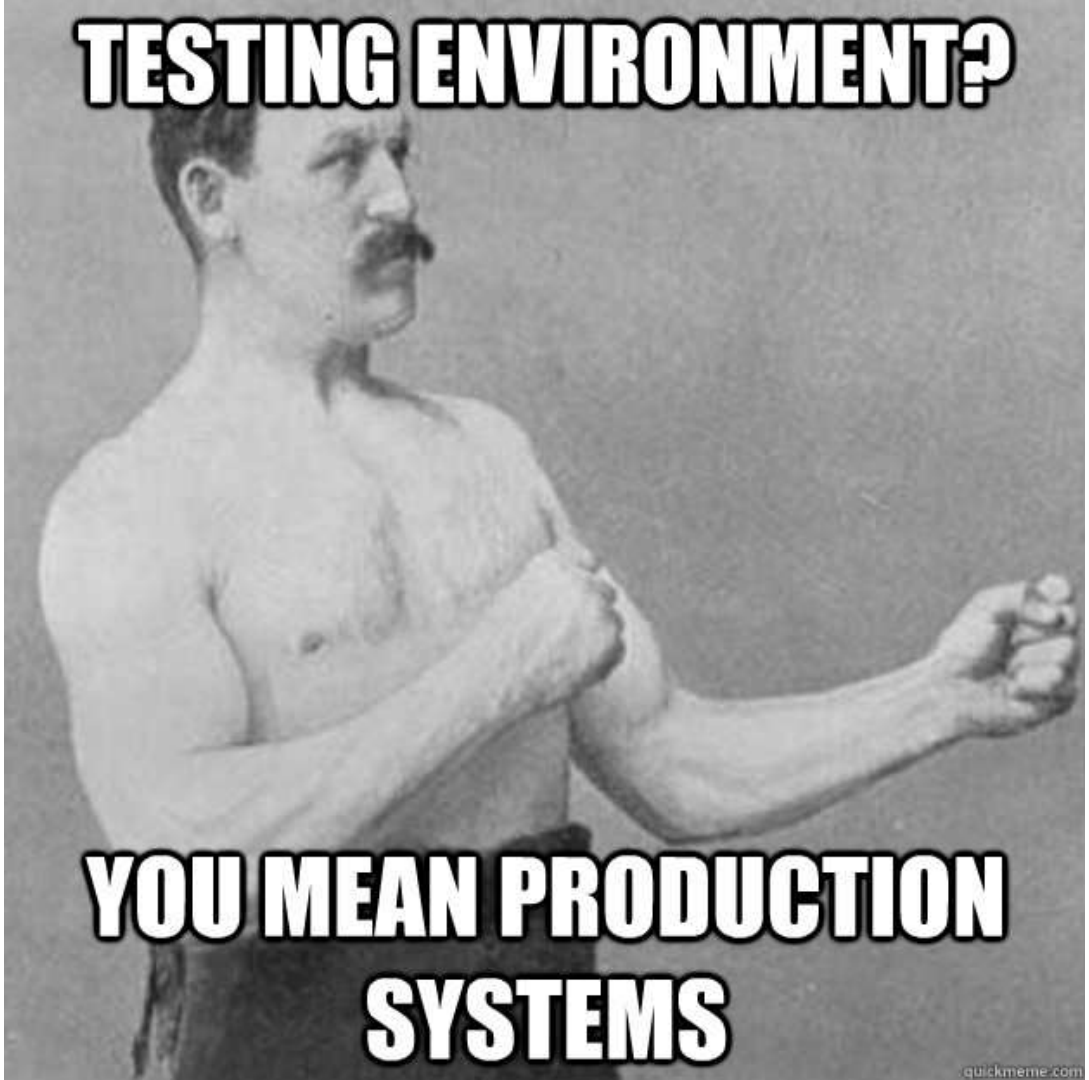
PYTHON
moduł unittest

```
from unittest import TestCase

class TestClass(TestCase):

    def test_test_a(self):
        …
        …
        self.assertEqual(a, b)
```

https://docs.python.org/3/library/unittest.html https://youtu.be/6tNS--WetLI

PYTHON
moduł unittest

```
assertEqual
assertAlmostEqual
assertNotEqual
assertIsInstance
assertTrue
assertFalse
assertIsNone
assertIn
```


TESTING ENVIRONMENT?
YOU MEAN PRODUCTION SYSTEMS
quickmeme.com

Thanks!!