Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 11: Testy jednostkowe, FIRST

Zasada FIRST

Zasady którymi należy się kierować podczas pisania testów.

Rozszyfrowanie akronimu:

- **F**ast (szybkie)
- **I**solated (izolowane)
- **R**epeatable (powtarzalne)
- **S**elf-Validation (jednoznaczne)
- **T**horough (dokładne)

Jakie powinny być nasze testy?

Fast (szybkie)

Testy powinny działać szybko, aby umożliwić częste uruchamianie.

Isolated (izolowane)

Testy nie powinny zależeć od siebie nawzajem ani od zewnętrznych czynników.

Repeatable (powtarzalne)

Testy powinny dawać te same wyniki za każdym razem, gdy są uruchamiane.

Self-validating (samowalidujące)

Testy powinny automatycznie określać, czy zakończyły się sukcesem, czy porażką.

Thorough (dokładne)

Testy powinny obejmować wszystkie istotne scenariusze.

Kiedy twój test będzie dokładny?

- Powinien obejmować wszystkie **happy path**.
- Spróbuj pokryć wszystkie skrajne przypadki.
- Wykonaj test pod względem niedozwolonych argumentów i zmiennych.
- Wykonaj test pod względem bezpieczeństwa i innych problemów.
- Wykonaj test dla dużych wartości.
- Test powinien uwzględniać każdy scenariusz, a nie tylko dążyć do **100%** pokrycia kodu.

Co to jest asercja?

- **Asercja** to instrukcja sprawdzająca, czy dany warunek jest prawdziwy.
- Jeśli warunek jest fałszywy, program zgłasza wyjątek `AssertionError`.
- Służą do wczesnego wykrywania błędów w kodzie.

```
assert x > 0, "x musi być dodatnie"
```

Framework unittest

Moduł **unittest** w Pythonie to wbudowany framework do testowania jednostkowego. Umożliwia pisanie testów, które automatycznie sprawdzają, czy poszczególne części kodu działają poprawnie.

Szkielet unittest

Minimalny kod potrzebny do uruchomienia testów.

```
import unittest

def add(a, b):
    return a + b

class TestAdd(unittest.TestCase):

    def test_add_numbers(self):
        self.assertEqual(add(2, 3), 5)

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

Dodatkowe metody, setUp() i tearDown()

```
import unittest

add = lambda a, b: a + b

class TestAddition(unittest.TestCase):

    def setUp(self):
        self.x, self.y = 2, 3

    def tearDown(self):
        self.x, self.y = None, None

    def test_add_numbers(self):
        self.assertEqual(add(self.x, self.y), 5)

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

Struktura testów (unittest)

- Klasa **TestCase**:
  - Podstawowy element struktury testów.
  - Każdy zestaw testów powinien dziedziczyć po `unittest.TestCase`.
  - Zawiera metody testowe, których nazwy zaczynają się od `test_`.
- Metody **setUp()** i **tearDown()**:
  - **setUp()**: wykonywane przed każdym testem.
  - **tearDown()**: wykonywane po każdym teście.
  - Służą do inicjalizacji i czyszczenia zasobów.

Omówienie wybranych asercji

assertEqual(a, b)

- Sprawdza, czy `a` jest równe `b`.
- Przydatna do porównywania wartości.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_equal_numbers(self):
        self.assertEqual(2 + 2, 4)
```

assertNotEqual(a, b)

- Sprawdza, czy `a` jest różne od `b`.
- Przydatna, gdy oczekujemy, że wartości będą różne.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_not_equal_numbers(self):
        self.assertNotEqual(2 + 2, 5)
```

assertTrue(x)

- Sprawdza, czy `x` jest prawdziwe.
- Używana do testowania warunków logicznych.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_true(self):
        self.assertTrue(3 > 1)
```

assertFalse(x)

- Sprawdza, czy `x` jest fałszywe.
- Używana do testowania warunków logicznych.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_false(self):
        self.assertFalse(1 > 3)
```

assertIs(a, b)

- Sprawdza, czy `a` i `b` to ten sam obiekt.
- Przydatna do sprawdzania tożsamości obiektów.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_same_object(self):
        list1 = [1, 2, 3]
        list2 = list1
        self.assertIs(list1, list2)
```

assertIsNot(a, b)

- Sprawdza, czy `a` i `b` to różne obiekty.
- Przydatna do sprawdzania, czy obiekty nie są identyczne.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_different_objects(self):
        list1 = [1, 2, 3]
        list2 = [1, 2, 3]
        self.assertIsNot(list1, list2)
```

assertIsNone(x)

- Sprawdza, czy `x` to `None`.
- Przydatna do sprawdzania, czy zmienna jest pusta.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_none(self):
        variable = None
        self.assertIsNone(variable)
```

assertIsNotNone(x)

- Sprawdza, czy `x` nie jest None.
- Przydatna do sprawdzania, czy zmienna ma wartość.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_not_none(self):
        variable = 5
        self.assertIsNotNone(variable)
```

assertIn(a, b)

- Sprawdza, czy `a` znajduje się w `b`.
- Przydatna do sprawdzania, czy element jest w liście, krotce lub słowniku.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_element_in_list(self):
        list_ = [1, 2, 3]
        self.assertIn(2, list_)
```

assertNotIn(a, b)

- Sprawdza, czy `a` nie znajduje się w `b`.
- Przydatna do sprawdzania, czy elementu nie ma w kolekcji.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_element_not_in_list(self):
        list_ = [1, 2, 3]
        self.assertNotIn(4, list_)
```

assertIsInstance(a, b)

- Sprawdza, czy `a` jest instancją klasy `b`.
- Przydatna do sprawdzania typu obiektu.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_instance_of_list(self):
        list_ = [1, 2, 3]
        self.assertIsInstance(list_, list)
```

assertNotIsInstance(a, b)

- Sprawdza, czy `a` nie jest instancją klasy `b`.
- Przydatna do sprawdzania, czy obiekt nie jest danego typu.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_not_instance_of_string(self):
        list_ = [1, 2, 3]
        self.assertNotIsInstance(list_, str)
```

assertRaises(exception, callable, *args, **kwds)

- Sprawdza, czy wywołanie `callable` zgłasza wyjątek `exception`.
- Przydatna do testowania obsługi błędów.

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_exception(self):
        with self.assertRaises(ValueError):
            int("abc")
```

Wzorce w testach

Given-When-Then & Arrange-Act-Assert

Given-When-Then

> GWT to wzorzec, który opisuje scenariusze testowe w sposób zbliżony do języka naturalnego.

- **Given** (Zakładając): Opisuje stan początkowy lub kontekst testu.
- **When** (Kiedy): Opisuje akcję lub zdarzenie, które jest testowane.
- **Then** (Wtedy): Opisuje oczekiwane wyniki lub zachowanie.

Given-When-Then (implementacja)

```python
# Given
list_ = [1, 2, 3]

# When
list_.append(4)

# Then
assert len(list_) == 4
```

Arrange-Act-Assert

> AAA to wzorzec, który dzieli test na trzy wyraźne sekcje.

- **Arrange** (Przygotowanie): W tej sekcji konfiguruje się dane testowe i stan początkowy.
- **Act** (Wykonanie): W tej sekcji wykonuje się testowaną operację.
- **Assert** (Sprawdzenie): W tej sekcji sprawdza się, czy wynik testu jest zgodny z oczekiwaniami.

Arrange-Act-Assert (implementacja)

```
# Arrange
x = 5
y = 10

# Act
result = x + y

# Assert
assert result == 15
```

GWT vs AAA

- Chociaż oba wzorce mają na celu organizację testów, GWT kładzie większy nacisk na opisanie zachowania systemu w sposób zrozumiały dla użytkownika, podczas gdy AAA skupia się na technicznym podziale testu na etapy.
- W praktyce, AAA i GWT często się przenikają, a wybór wzorca zależy od kontekstu i preferencji zespołu.

Rozwiązanie zadania z poprzedniego wykładu

> Funkcja obliczająca cenę biletów dla grupy osób.

Oblicz cenę biletów dla wybranej ilości osób. Jeśli wśród nich są dzieci, należy obniżyć cenę ich biletów o **50%**.

Dane wejściowe:

- **tickets** - liczba biletów do zakupu.
- **children** - liczba dzieci.
- **regular_price** - cena regularna.

Rozwiązanie zadania (implementacja)

```
def calculate_tickets_price(tickets, children, regular_price):
    if tickets < 0 or children < 0 or regular_price < 0:
        raise ValueError("Wartości nie mogą być ujemne.")
    if tickets < children:
        raise ValueError("Liczba biletów musi być większa lub równa liczbie dzieci.")

    total_price = tickets * regular_price
    discount = 0.0

    if children > 0:
        discount = 0.5 * regular_price * children
        total_price -= discount

    return {"total_price": max(0.0, total_price), "discount": discount}
```

Rozwiązanie zadania (testy, part 1)

```
class TestCalculateTicketsPrice(unittest.TestCase):

    def test_regular_price(self):
        result = calculate_tickets_price(5, 0, 6)
        self.assertEqual(result["total_price"], 30.0)
        self.assertEqual(result["discount"], 0.0)

    def test_discounted_price(self):
        result = calculate_tickets_price(5, 2, 6)
        self.assertEqual(result["total_price"], 24.0)
        self.assertEqual(result["discount"], 6.0)
```

Rozwiązanie zadania (testy, part 2)

```
class TestCalculateTicketsPrice(unittest.TestCase):
    def test_zero_tickets(self):
        with self.assertRaises(ValueError):
            calculate_tickets_price(0, 2, 6)

    def test_negative_tickets(self):
        with self.assertRaises(ValueError):
            calculate_tickets_price(-5, 2, 6)

    def test_invalid_tickets_children_ratio(self):
        with self.assertRaises(ValueError):
            calculate_tickets_price(1, 2, 6)
```

Rozwiązanie zadania (uruchomienie)

```
.....
----------------------
Ran 5 tests in 0.000s
OK
```

Koniec