

Testowanie aplikacji

Poziomy testowania, unittest

Poziomy testowania

Poziom testów czyli grupa czynności testowych, które są razem zorganizowane.

Istotą rozróżniania poziomów testów jest to, że każdy poziom ma inne cele testowania, ma zwykle inną podstawę testów, a także inny obiekt testowania.

Typowe poziomy testowania:

- **jednostkowe** - *testowanie pojedynczych modułów,*
- **integracyjne** - *testowanie wykonywane w celu wykrycia defektów podczas interakcji między komponentami lub systemami,*
- **systemowe** - *testowanie zintegrowanego systemu w celu sprawdzenia jego zgodności z wyspecyfikowanymi wymaganiami,*
- **akceptacyjne** - *testowanie formalnie przeprowadzane w celu umożliwienia użytkownikowi, klientowi lub innemu ustalonemu podmiotowi ustalenia, czy zaakceptować system lub moduł.*



Unittest

*The unittest unit testing framework was originally inspired by **JUnit** and has a similar flavor as major unit testing frameworks in other languages.* (źródło: <https://docs.python.org/3/library/unittest.html>)

unittest jest biblioteką dostarczaną razem z Pythonem, nie wymaga instalacji.

Kluczowe słownictwo:

- *test fixture* - określa akcje potrzebne do przygotowania testu i jego zakończenia.
- *test case* - przypadek testowy, test.
- *test suite* - kolekcja test case i test suite, służy do grupowania testów, które należy wykonać razem.



Unittest - Struktura projektu

Przykładowa struktura plików projektu.

Pliki z testami umieszczamy w katalogu z implementacją lub w osobnym katalogu o nazwie tests.

Ważne jest, by nazwa pliku z testami zaczynała się od frazy `test_`, inaczej testy się nie uruchomią!



Testowanie aplikacji

**struktura plików:
testy w katalogu
z implementacją**

```
.  
|-- kalkulator.py  
'-- utils  
    |-- strings.py  
    |-- test_strings.py  
    |-- user.py  
    '-- test_user.py
```

**struktura plików:
testy w osobnym
katalogu**

```
.  
|-- kalkulator.py  
|-- utils  
|   |-- strings.py  
|   '-- user.py  
'-- tests  
    |-- __init__.py  
    '-- utils  
        |-- __init__.py  
        |-- test_strings.py  
        '-- test_user.py
```

wymagany plik

nazwa pliku zaczyna się od `test_`

Unittest - Tworzenie testów

Tworzenie testów

rozpoczynamy od utworzenia klasy, w której umieszczone będą poszczególne przypadki testowe.

Klasa ta musi dziedziczyć po `unittest.TestCase`.

Nazwy poszczególnych testów powinny zaczynać się od frazy `test_`, inaczej testy się nie uruchomią!

```
tests/utills/test_strings.py:  
import unittest  
  
from utills.strings import czy_liczba  
  
class CzyLiczba(unittest.TestCase):  
  
    def test_poprawna_liczba_dodatnia(self):  
        result = czy_liczba("5")  
        self.assertTrue(result)
```

pamiętamy o zaimportowaniu biblioteki

konieczne dziedziczenie

nazwa zaczyna się od `test_`



Unittest - Warunki testowe

Warunki testowe

sprawdzone są za pomocą metod typu `assert*`.

W przykładzie została użyta metoda **`assertTrue()`**, sprawdza ona czy podany argument ma wartość `True`, jeśli nie, wykonywany test zostanie przerwany i pojawi się komunikat o błędzie.

`tests/utills/test_strings.py:`

```
import unittest
```

```
from utills.strings import czy_liczba
```

```
class CzyLiczba(unittest.TestCase):
```

```
    def test_poprawna_liczba_dodatnia(self):  
        result = czy_liczba("5")  
        self.assertTrue(result)
```



Unittest - Warunki testowe

Wybrane metody typu `assert*()`:

- `assertEqual(a, b)` *a == b*
- `assertNotEqual(a, b)` *a != b*
- `assertTrue(x)` *bool(x) is True*
- `assertFalse(x)` *bool(x) is False*
- `assertIs(a, b)` *a is b*
- `assertIsNot(a, b)` *a is not b*
- `assertIsNone(x)` *x is None*
- `assertIsNotNone(x)` *x is not None*
- `assertIn(a, b)` *a in b*
- `assertNotIn(a, b)` *a not in b*



Unittest - Warunki testowe

Wybrane metody typu `assert*()`:

- `assertAlmostEqual(a, b)` `round(a-b, 7) == 0`
- `assertNotAlmostEqual(a, b)` `round(a-b, 7) != 0`
- `assertGreater(a, b)` `a > b`
- `assertGreaterEqual(a, b)` `a >= b`
- `assertLess(a, b)` `a < b`
- `assertLessEqual(a, b)` `a <= b`



Unittest - Warunki testowe

W metodach typu **assert***() za parametr **a** podajemy wartość uzyskaną w wyniku działania testu, parametr **b** powinien przyjmować wartość oczekiwaną. Ma to znaczenie podczas wyświetlania komunikatu o błędzie.

W przypadku, gdy warunek metody typu **assert***() nie zostanie spełniony test zostaje przerwany i oznaczony jako FAIL.



Unittest - Warunki testowe

Metoda **assertRaises()**, sprawdza czy instrukcje umieszczone w specjalnym bloku **with** rzucają wyjątek, jako parametr przyjmuje klasę oczekiwanego wyjątku.

```
import unittest

class CheckDigit(unittest.TestCase):

    def test_liczba_z_litera(self):
        with self.assertRaises(NotDigit):
            check_digit("3a")
```



Testowanie aplikacji

<https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises>

Unittest - setUp, tearDown

Testy mogą również wykorzystywać specjalne metody `setUp()` oraz `tearDown()` do przygotowania i zakończenia testu.

Metody te zostaną wywołane przed rozpoczęciem i po zakończeniu każdego z testów.

```
import unittest

class TestKalkulator(unittest.TestCase):

    def setUp(self):
        self.kalkulator = Kalkulator()

    def test_dodaj(self):
        result = self.kalkulator.dodaj(2, 3)
        self.assertEqual(result, 5)

    def tearDown(self):
        self.kalkulator = None
```



Unittest - Uruchamianie testów

Testy uruchamiamy za pomocą polecenia **python -m unittest** wykonywanego z poziomu katalogu głównego projektu.

Argumentem do polecenia jest:

- ścieżka w formie importu,
- ścieżka do pliku z testem,
- polecenie `discover` - automatycznie znajduje testy.



Unittest - Uruchamianie testów

Przykład uruchomienia
z argumentem:

- **ścieżki w formie importu,**
- ścieżki do pliku z testem,
- poleceniem `discover`.

```
$ python -m unittest tests.utils.test_strings
```

```
.
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```



Testowanie aplikacji

Unittest - Uruchamianie testów

Przykład uruchomienia
z argumentem:

- ścieżki w formie importu,
- **ścieżki do pliku z testem,**
- poleceniem `discover`.

```
$ python -m unittest tests/utils/test_strings.py
.  
-----  
Ran 1 test in 0.000s  
  
OK
```



Testowanie aplikacji

Unittest - Uruchamianie testów

Przykład uruchomienia
z argumentem:

- ścieżki w formie importu,
- ścieżki do pliku z testem,
- **poleceniem discover.**

```
$ python -m unittest discover
```

```
.
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```



Unittest - Uruchamianie testów

Warto wiedzieć, że polecenia te przyjmują dodatkowy argument -v, który powoduje wyświetlanie większej ilości informacji.

```
$ python -m unittest discover -v
test_poprawna_liczba_dodatnia \
    (tests.utils.test_strings.CzyLiczba) ... ok
.
-----
Ran 1 test in 0.000s
OK
```

```
$ python -m unittest -v tests/utils/test_strings.py
test_poprawna_liczba_dodatnia \
    (tests.utils.test_strings.CzyLiczba) ... ok
.
-----
Ran 1 test in 0.000s
OK
```



Testowanie aplikacji

Unittest - Uruchamianie testów

Przykładowy wydruk
z wykonania testu,
który się nie powiódł.

```
$ python -m unittest discover
=====
FAIL: test_poprawna_liczba_dodatnia \
      (tests.utils.test_strings.CzyLiczba)
-----
Traceback (most recent call last):
  File "tests/utils/test_strings.py", line 10, \
        in test_poprawna_liczba_dodatnia
    self.assertTrue(result)
AssertionError: False is not true

-----

Ran 1 test in 0.000s
FAILED (failures=1)
```



Testowanie aplikacji

Pytania

1. Jaki przedrostek powinny zawierać nazwy plików z testami i nazwy testów?
2. Jakich funkcji używa unittest do sprawdzenia warunków testowych?
3. Co się stanie jeśli warunek funkcji typu `assert*()` nie zostanie spełniony?



Literatura

1. unittest — Unit testing framework,
<https://docs.python.org/3/library/unittest.html>
2. Adam Roman, Testowanie i jakość oprogramowania. Metody, narzędzia, techniki, 2015,
<https://ksiegarnia.pwn.pl/Testowanie-i-jakosc-oprogramowania.-Modele-techniki-narzedzia.,732463348,p.html>



Testowanie aplikacji

