TESTOWANIE:
TESTY JEDNOSTKOWE, UŻYCIE DEBUGGERA
WSTĘP DO TDD

Plan szkolenia

Plan modułu

1. **TESTOWANIE**
   - wprowadzenie
   - moduł *pytest*
   - moduł *unittest*
   - TDD

TEORIA

Poziomy testowania

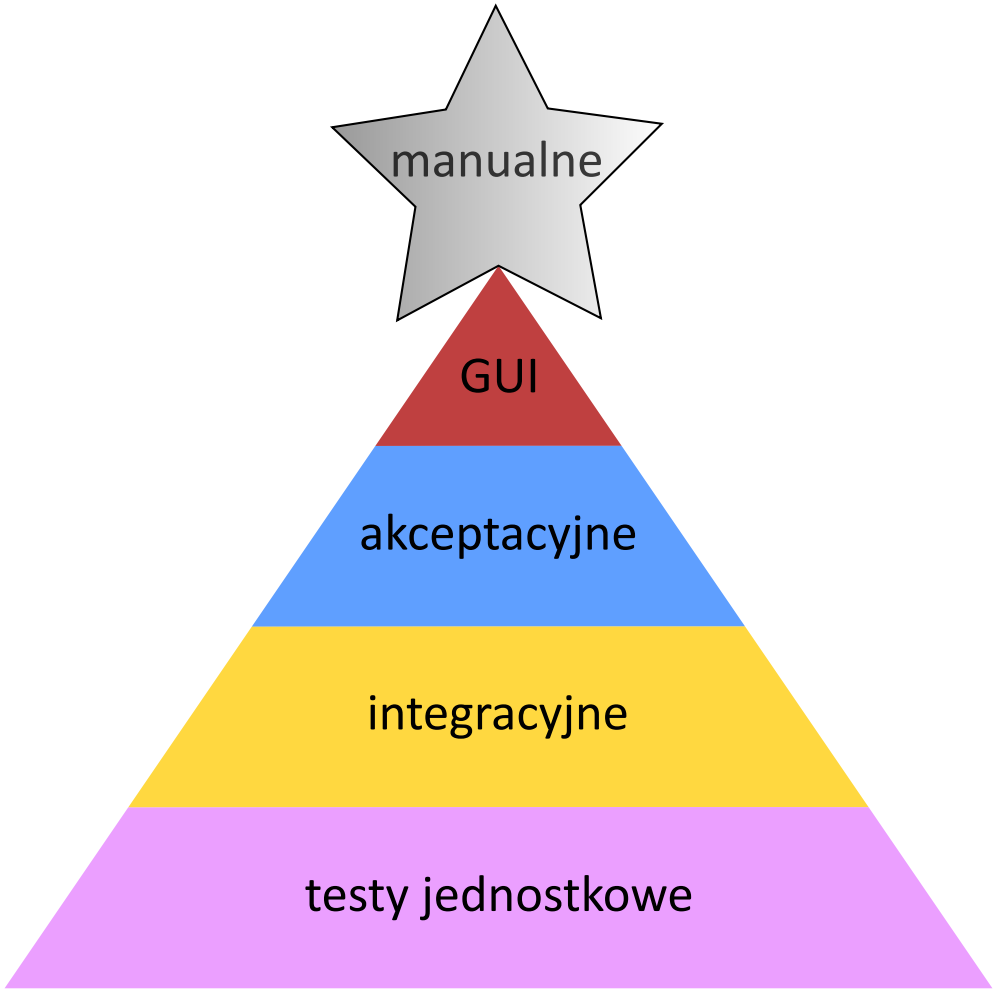
- Poziomy testowania:

jednostkowy	czy obiekty robią co trzeba i czy wygodnie się nimi pracuje?
integracyjny	czy warstwy/moduły współpracują ze sobą poprawnie?
akceptacyjny	czy cały system działa wg założeń?

Testy jednostkowe

- **Testy jednostkowe** (*unit tests*) weryfikują poprawność działania pojedynczego elementu
  - funkcje, metody
  - klasy, instancje
  - stany
- Testy jednostkowe powinny być pisane przez programistów!
- Są podstawą piramidy testów

Piramida testów



Testy jednostkowe

- Za pomocą testów jednostkowych staramy się zweryfikować funkcjonalność aplikacji na najbardziej podstawowym poziomie
- Testujemy każdą jednostkę kodu, zazwyczaj metodę, w izolacji od innych, aby sprawdzić, czy w określonych warunkach reaguje w oczekiwany sposób
- Przeniesienie testowania na ten poziom daje pewność, że każda część aplikacji będzie zachowywać się zgodnie z oczekiwaniami i umożliwia wykrycie przypadków brzegowych, w których aplikacja może działać w niestandardowy sposób i odpowiednio radzić sobie z nimi

Jak napisać dobry test?

- Testuj operacje, które nie powinny się udać
- Testuj operacje, które powinny się udać
- Należy pamiętać o wartościach specjalnych
- . . . i nie zapomnieć o wartościach brzegowych
- Testuj wartości o +/-1 od wartości brzegowych
- Testuj wartości standardowe
- Testuj wyjątki
- Testuj nadmiarowe argumenty
- Użyj zdecydowanie za mało i za dużo danych wejściowych
- Testuj bloki kodu warunkowego

pytest

- Dokumentacja modułu ☞ *pytest*
- Darmowy
- Bardzo szybki
- Bardzo Pythonic
- Możliwość rozproszenia
- Dostępne pluginy
- Naturalne asercje

pytest

Przykład

```
import pytest

def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5

def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

pytest

Przykład – setup/teardown

```
class TestClass(object):
    @classmethod
    def setup_class(cls):
        pass

    def teardown_method(self, method):
        pass

    def test_case(self):
        pass
```

pytest

Przykład

```
import sys
import pytest

minversion = pytest.mark.skipif(sys.version_info >= (3, 3),
                                reason="requires python3.3")

@minversion
def test_func():
    pass

@pytest.mark.xfail   # Spodziewamy się niepowodzenia
def test_func2():
    pass
```

pytest

Generator testów

```
def pytest_generate_tests(metafunc):
    if "numiter" in metafunc.funcargnames:
        metafunc.parametrize("numiter", range(10))

def test_func(numiter):
    assert numiter < 9
```

pytest

Generator testów

```
========================= test session starts ==============================
platform darwin -- Python 2.7.5 -- pytest-2.4.2
collected 10 items

test_g.py .........F

============================== FAILURES ====================================
_______________________________ test_fu[9] ____________________________________

numiter = 9

    def test_fu(numiter)
>       assert numiter < 9
E       assert 9 < 9

test_g.py:6: AssertionError
====================== 1 failed, 9 passed in 0.03 seconds ==================
```

Biblioteka *unittest*

- Biblioteka *unittest* – wbudowana biblioteka służąca do automatyzacji testów jednostkowych w Pythonie, wzorowana na *JUnit*, o podobnych możliwościach jak frameworki testów jednostkowych w innych językach
- Biblioteka umożliwia:
  - automatyzację testów
  - współdzielenie kodu konfiguracji i kończenia testów
  - grupowanie testów w zestawy
  - niezależność testów od frameworka raportowania
- Pełna dokumentacja jest dostępna ☞ tutaj

Podstawowe pojęcia

klasa przypadków testowych (*test case class*)	klasa bazowa dla wszystkich klas w modułach testowych wszystkie klasy testowe są wyprowadzane z tej klasy
środowisko testowe (*test fixture*)	funkcje lub metody wykonywane przed i po blokach kodu testowego, konieczne do ich wykonania oraz wszelkie powiązane z nimi działania "czyszczące" (przywracające stan pierwotny)
asercje (*assertions*)	funkcje lub metody używane do weryfikacji zachowania testowanego komponentu
zestaw testów (*test suite*)	zbiór powiązanych przypadków testowych lub zestawów testów służy do grupowania testów, które powinny być wykonywane razem
przypadek testowy (*test case*)	indywidualna jednostka testowa – w *unittest* jest nim pojedyncza metoda sprawdza odpowiedź na określony zestaw danych wejściowych
moduł uruchamiający (*test runner*)	program lub fragment kodu, który wykonuje zestaw testów i dostarcza wyniki użytkownikowi
formater wyników testów (*test result formatter*)	formatuje wyniki wykonanych testów do wybranego, czytelnego dla człowieka formatu (np. zwykły tekst, HTML, XML, . . .)

Klasa testowa

- Podstawowymi elementami składowymi testów jednostkowych są przypadki testowe – pojedyncze scenariusze, które należy skonfigurować i sprawdzić pod kątem poprawności
- Przypadki testowe są reprezentowane przez instancje *TestCase*
- Aby tworzyć własne przypadki testowe, należy rozszerzyć klasę *TestCase* (tworząc **klasę testową**) lub użyć *FunctionTestCase*
- Nazwy metod testowych mają przedrostek *test_*
- Metody testowe są wykonywane w porządku alfabetycznym, niezależnie od kolejności ich umieszczenia w kodzie

Moduły testowe

- W jednym pliku można umieścić wiele klas testowych
- Taki plik nosi nazwę **modułu testowego**
- Wszystkie klasy testowe w module są wykonywane w porządku alfabetycznym

Środowisko testowe

- **Środowisko testowe** (*test fixture*) to zestaw czynności wykonywanych **przed** i **po** testach
- Są one implementowane jako metody klasy *TestCase* i mogą być nadpisane do własnych celów

	WYKONYWANE...
FUNKCJE: *setUpModule()* *tearDownModule()*	 **przed** jakąkolwiek metodą w module testów **po** wszystkich metodach w module testów
METODY KLASOWE: *setUpClass(cls)* *tearDownClass(cls)*	 **przed** jakąkolwiek metodą w klasie testów **po** wszystkich metodach w klasie testów
METODY INSTANCYJNE: *setUp(self)* *tearDown(self)*	 **przed** każdą metodą w klasie testów **po** każdej metodzie w klasie testów

Klasa testowa

Przykład

```
import unittest

def setUpModule():
    print('setUpModule function')

def tearDownModule():
    print('tearDownModule function')

class KlasaTestowa(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        print('setUpClass class method')

    @classmethod
    def tearDownClass(cls):
        print('\ntearDownClass class method')
```

Klasa testowa

Przykład cd.

```
    # cd...
    def setUp(self):
        print('\nsetUp instance method')

    def tearDown(self):
        print('tearDown instance method')

    def test_if_uppercase(self):
        self.assertTrue("TEST".isupper())
        print('test_if_uppercase instance method')

    def test_if_not_uppercase(self):
        self.assertFalse("test".isupper())
        print('test_if_not_uppercase instance method')

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

Klasa testowa

Wynik testów

```
setUpModule function
setUpClass class method

setUp instance method
test_if_not_uppercase instance method
tearDown instance method

setUp instance method
test_if_uppercase instance method
tearDown instance method

tearDownClass class method
tearDownModule function
..
----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK
```

Przypadki testowe

- Do identyfikacji przypadków testowych mogą być przydatne metody:

Użyteczne metody

```
id()
shortDescription()
```

id	zwraca tekst specyficzny dla danego przypadku testowego zwykle jest to pełna nazwa przypadku testowego, zawierająca nazwę modułu, klasy i metody testowej
shortDescription	opis przypadku testowego zwykle to pierwsza linia dokumentacji (docstring'a)

Asercje

- W metodach testowych wykorzystuje się **asercje**
- Jeżeli argument **spełnia warunek asercji**, to test kończy się pomyślnie, w przeciwnym razie zawodzi

- Można też wymusić **bezwarunkowe** fiasko testów:

Fiasko testów

```
fail(msg)
```

Asercje

METODA	SPRAWDZA, CZY...
assertEqual(*a*, *b*) **assertNotEqual**(*a*, *b*)	*a* == *b* *a* != *b*
assertTrue(*x*) **assertFalse**(*x*)	bool(*x*) == True bool(*x*) == False
assertIs(*a*, *b*) **assertIsNot**(*a*, *b*)	*a* is *b* *a* is not *b*
assertIsNone(*x*) **assertIsNotNone**(*x*)	*x* is None *x* is not None
assertIn(*a*, *b*) **assertNotIn**(*a*, *b*)	*a* in *b* *a* not in *b*
assertIsInstance(*a*, *b*) **assertINotIsInstance**(*a*, *b*)	isInstance(*a*, *b*) not isInstance(*a*, *b*)

- Wszystkie metody asercji mogą przyjąć dodatkowy parametr tekstowy, pełniący rolę komunikatu błędu, jeżeli asercja zakończy się fiaskiem

Asercje

- Metody asercji dokonujące bardziej specyficznych porównań

METODA	SPRAWDZA, CZY...
assertAlmostEqual(*a*, *b*) **assertNotAlmostEqual**(*a*, *b*)	round(*a* - *b*, 7) == 0 round(*a* - *b*, 7) != 0
assertGreater(*a*, *b*) **assertGreaterEqual**(*a*, *b*)	*a* > *b* *a* >= *b*
assertLess(*a*, *b*) **assertLessEqual**(*a*, *b*)	*a* < *b* *a* <= *b*
assertRegex(*s*, *r*) **assertNotRegex**(*s*, *r*)	*r*.search(*s*) not *r*.search(*s*)
assertCountEqual(*a*, *b*)	*a* i *b* zawierają te same elementy, w tej samej liczbie, niezależnie od kolejności

Asercje

- Poniższe metody asercji służą do porównywania wybranych typów kontenerów
- Są wykorzystywane automatycznie przez metodę *assertEqual* – stąd w praktyce nie ma potrzeby wywoływania ich samodzielnie

METODA	SŁUŻY DO PORÓWNANIA...
assertMultiLineEqual(*a*, *b*)	tekstów
assertSequenceEqual(*a*, *b*)	sekwencji
assertListEqual(*a*, *b*)	list
assertTupleEqual(*a*, *b*)	krotek
assertSetEqual(*a*, *b*)	zbiorów (typu *set* lub *frozenset*)
assertDictEqual(*a*, *b*)	słowników

Pomijanie testów i oczekiwanie fiaska

- Do pominięcia testów oraz zdefiniowania testu w którym spodziewamy się fiaska można użyć dekoratorów
- Dekorator zastosowany na poziomie metody dotyczy tylko tego przypadku testowego, zaś zastosowany na poziomie klasy przypadków testowych dotyczy wszystkich przypadków w tej klasie

@**skip**(*reason*)	bezwarunkowe pominięcie udekorowanego testu parametr *reason* opisuje powód pominięcia testu
@**skipIf**(*condition*, *reason*)	pominięcie udekorowanego testu, o ile podany warunek jest spełniony
@**skipUnless**(*condition*, *reason*)	pominięcie udekorowanego testu, o ile podany warunek nie jest spełniony
@**expectedFailure**	wskazanie, że oczekiwane jest fiasko testu

Wyjątki w przypadkach testowych

- Wystąpienie nieobsłużonego wyjątku w przypadku testowym, powoduje zakończenie testu fiaskiem
- Można też za pomocą asercji sprawdzać możliwość wystąpienia wyjątku

Oczekiwanie wystąpienia wyjątku

```
assertRaises(exception, callable, *args, **kwds)
assertRaises(exception, *, msg=None)
```

- W drugim wariancie zwracany jest menedżer kontekstu
- Ta wersja umożliwia podanie komunikatu błędu

Wyjątki w przypadkach testowych

Oczekiwanie na zgłoszenie wyjątku

```
import unittest

def f():
    raise RuntimeError()

class Tests(unittest.TestCase):
    def test_a(self):
        self.assertRaises(RuntimeError, f)

    def test_b(self):
        with self.assertRaises(RuntimeError) as ctx:
            f()  # można wstawić kod, zamiast wywołania funkcji
        exc = ctx.exception
        # tu można dokonać dodatkowych weryfikacji odnośnie
        # otrzymanego wyjątku
        self.assertEqual(exc.__class__.__name__, 'RuntimeError')
```

Wykonywanie testów

- Do uruchomienia modułu testowego wykorzystuje się metodę *main*
- Można jej przekazać argument *verbosity* kontrolujący "szczegółowość" prezentowanych wyników testów

Inne biblioteki testowe

- Istnieje też wiele innych bibliotek testowych, np.:

doctest	opis dostępny jest ☞ tutaj
nose	opis dostępny jest ☞ tutaj
nose2	opis dostępny jest ☞ tutaj

TDD

- **Rozwój sterowany testami** (*Test-Driven Development*, w skrócie *TDD*)
- Proces polegający na powtarzaniu krótkich cyklów rozwojowych
- Cykl składa się z kilku etapów
- Nie wolno pominąć zasady: **najpierw test, później kod**
- Nie, to nie znaczy, że jesteś testerem, a nie deweloperem
- Od teraz jesteś prawdziwym hakerem!
- Od teraz zawsze będziemy chcieli programować wg TDD!

Etap I – napisanie testu

- Dla ulepszenia aktualnej funkcjonalności
- Dla nowej funkcjonalności
- Pisanie dobrych testów wymaga większego doświadczenia, niż pisanie dobrego kodu
- Zanim napiszemy test, należy w pełni zrozumieć problem, który chcemy testować

Etap I – napisanie testu

- Dobry test powinien zastąpić dokumentację techniczną – kolejny zysk czasowy!
- *Code review* to w 90% oglądanie testów
- Pojedynczy unit test powinien weryfikować elementarną funkcjonalność
- Nie testujemy implementacji kompilatora, linkera, parsera, czy maszyny

Etap I – napisanie testu

- Testowanie funkcji *sqrt()*
  - należy koniecznie przetestować dla wartości argumentu: 0, 1, 49,100
  - *sqrt*(0.25) = 0.5 ?
  - czy *sqrt*(7) * *sqrt*(7) = (6.99999; 7.00001) ?
  - wartości ujemne
  - wartość *None*
  - *sqrt*(10 ** 100) ?
  - referencja funkcji?

Etap II – niepowodzenie nowych testów

- Uruchomienie TestSuite
- Stare testy powinny przechodzić tak, jak do tej pory
- Nowe testy powinny zgłaszać niepowodzenie

Etap III – implementacja

- Napisz fragment kodu
- Najlepiej tak, aby testy kończyły się sukcesem :-)
- Kod moze być niedoskonały
- Kiepski styl i różne sztuczki dozwolone
- Najważniejsze, aby testy przechodziły! :-)
- *“Fake it till you make it”*
- Nie należy pisać kodu, którego nie obejmują przygotowane testy

Etap IV – uruchomienie TestSuite

- Dopóki wszystkie testy nie przejdą, wracamy do etapu III

Etap V – refaktoryzacja

- Popraw kod tak, aby był dobry
- Popraw kod tak, aby był elegancki
- Popraw kod tak, aby przeszedł *code review*
- Popraw kod tak, aby nie miał zbędnych śmieci
- W każdym momencie możesz wrócić do etapu IV

Refaktoryzacja

- Technika zmiany wewnętrznej struktury kodu bez modyfikacji jego zachowania
- Funkcjonalność pozostaje taka sama
- Bazuje na serii przekształceń zachowujących semantykę kodu
- Testy nie powinny być modyfikowane w procesie refaktoryzacji

Cel refaktoryzacji

- **Nie jest** celem nowa funkcjonalność
- Podniesienie jakości wytwarzanego oprogramowania
- Czytelność
- Porządek

Cel refaktoryzacji

- Optymalizacja
- Eliminacja tzw. *bottlenecks*
- Ograniczenie redundancji kodu
- Ograniczenie złożoności kodu
- Podniesienie *maintainability*

Dziękujemy za uwagę