

Kurs języka Python

Wykład 11.

Marcin Młotkowski

4 stycznia 2010

- 1 Kontrola poprawności podczas biegu programu
- 2 Testowanie oprogramowania
- 3 Pomiar wydajności aplikacji
- 4 Debuggowanie

Asercje

- Asercja to formuła logiczna;
- Asercji używa się do kontrolowania czy np. wartość zmiennej ma odpowiedni typ lub mieści się w pożądanym zakresie;
- Do kontroli używa się instrukcji
`assert wyrażenie`
- W przypadku niespełnienia wyrażenia zgłaszany jest wyjątek `AssertionError`

Przykład użycia asercji

```
def dodaj(x, y):  
    assert type(x) == type(1)  
    assert type(y) == type(1)  
    return x + y
```

```
>>> dodaj('dwa', 'dwa')  
Traceback (most recent call last):  
  File "asercje.py", line 14, in <module>  
    print dodaj("dwa", "dwa")  
  File "asercje.py", line 6, in dodaj  
    assert type(x) == type(1)  
AssertionError
```

Przykład użycia asercji

```
def dodaj(x, y):  
    assert type(x) == type(1)  
    assert type(y) == type(1)  
    return x + y
```

```
>>> dodaj('dwa', 'dwa')  
Traceback (most recent call last):  
  File "asercje.py", line 14, in <module>  
    print dodaj("dwa", "dwa")  
  File "asercje.py", line 6, in dodaj  
    assert type(x) == type(1)  
AssertionError
```

Uwagi

Asercje spowalniają działanie programu.

Wyłączanie asercji

- Asercje są sprawdzane w zależności od zmiennej logicznej `__debug__`;
- domyślna wartość: **True**;
- zmiennej `__debug__` nie można modyfikować w czasie wykonywania programu
- W przypadku uruchomienia programu z opcją `'-O'` (optymalizacja) wartością `__debug__` jest **False**.

Inne wykorzystanie `__debug__`

```
if __debug__:  
    print dodaj(2,2)  
    print dodaj("dwa", "dwa")
```

Wprowadzenie do testowania

Test jednostkowy(ang. unit test)

Test sprawdzający poprawność pojedynczego elementu oprogramowania: metody, klasy czy procedury.

Zestaw testów (ang. test suite)

Implementuje się zestawy testów, które można uruchomić automatycznie.

Testy regresyjne

Testy przeprowadzane po wprowadzeniu zmian do dobrze działającego kodu.

Wprowadzenie do testowania

Test jednostkowy(ang. unit test)

Test sprawdzający poprawność pojedynczego elementu oprogramowania: metody, klasy czy procedury.

Zestaw testów (ang. test suite)

Implementuje się zestawy testów, które można uruchomić automatycznie.

Testy regresyjne

Testy przeprowadzane po wprowadzeniu zmian do dobrze działającego kodu.

Wprowadzenie do testowania

Test jednostkowy(ang. unit test)

Test sprawdzający poprawność pojedynczego elementu oprogramowania: metody, klasy czy procedury.

Zestaw testów (ang. test suite)

Implementuje się zestawy testów, które można uruchomić automatycznie.

Testy regresyjne

Testy przeprowadzane po wprowadzeniu zmian do dobrze działającego kodu.

Testy jednostkowe

Testy jednostkowe można traktować jako specyfikację klasy bądź modułu.

Testowanie

Co jest testowane:

- czy poprawne dane dają poprawny wynik;
- czy niepoprawne dane dają oczekiwany (np. niepoprawny wynik) lub wyjątek.

Narzędzia do testowania w Pythonie

- PyUnit
- PyDoc

Zadanie

Napisać funkcję `deCapitalize` z argumentem typu `string` i zwracającą `string`

- Unifikacja imienia i nazwiska do postaci 'Imie Nazwisko', np. `deCapitalize('JAN KOWALSKI')`: 'Jan Kowalski'
- Kontrola typu, gdy argument nie jest typu `string`, zgłaszany jest wyjątek `ArgumentNotStringError`

PyUnit

- Testy są zebrane w odrębnym pliku (plikach)
- Można wskazywać, jakie testy mają być wykonane

Implementacja

```
class ArgumentNotStringError(Exception): pass
```

```
def deCapitalize(nazwisko):
```

```
    """
```

```
    Zamiana napisów (imion i nazwisk) pisanych  
    wielkimi literami
```

```
    """
```

Unittest

Implementacja

```
import unittest
import types
class TestDeCapitalize(unittest.TestCase):
```

Zamiana na poprawną postać

```
znaneWartosci = [  
    ("jaN KoWaLski", "Jan Kowalski"),  
    ("cLaude leVi-StrAuSs", "Claude Levi-Strauss"),  
    ("JeRzy auGust MniSzEch", "Jerzy August Mniszech")  
]
```

```
class TestdeCapitalize
```

```
    def testProsty(self):  
        """Proste sprawdzenia"""  
        for zly, dobry in self.znaneWartosci:  
            res = deCapitalize(zly)  
            self.assertEqual(dobry, res)
```

Test na identyczność

definicja danych

```
listaNazwisk = [ "Benedykt Polak", "Fryderyk Joliot-Curie"]
```

```
def testIdent(self):  
    """Nie zamieni poprawnych nazwisk"""  
    for nazw in self.listaNazwisk:  
        self.assertEqual(nazw, deCapitalize(nazw),  
                           "Zmiana poprawnego nazwiska!")
```

Niepoprawne wyniki

```
psuj = [  
    ("SpYtko z MeLsztyrna", "Spytko z Melsztyna"),  
    ("SkaRbiMlr Z rodu AwDańców", "Skarbimir z rodu  
    Awdańców"),  
]
```

```
def testZly(self):  
    """Nie radzi sobie"""  
    for zly, dobry in self.psuj:  
        res = deCapitalize(zly)  
        self.assertNotEqual(dobry, res)
```

Przypomnienie

```
class ArgumentNotStringError(Exception): pass

def deCapitalize(nazwisko):
    """
    Zamiana napisów (imion i nazwisk) pisanych wielkimi literami
    """
    if type(nazwisko) != types.StringType:
        raise ArgumentNotStringError
```

Metoda testująca

```
def testDziedzina(self):
    self.assertRaises(ArgumentNotStringError, deCapitalize, 10)
```

Podsumowanie

testy.py

```
import unittest, types
import testowany_modul

class TestdeCapitalize(unittest.TestCase):
    def testProsty(self): ...
    def testIdent(self): ...
    def testDziedzina(self): ...

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

Uzupełnienie

Metoda `TestDeCapitalize.setUp(self)`

Inicjowanie wstępne wykonywane przed każdym testem (zakładanie baz danych i tabel, tworzenie plików/tabel z przykładowymi danymi).

Metoda `TestDeCapitalize.tearDown(self)`

sprzątanie wykonywane po każdym teście (usuwanie tymczasowych plików etc).

Uzupełnienie

Metoda `TestDeCapitalize.setUp(self)`

Inicjowanie wstępne wykonywane przed każdym testem (zakładanie baz danych i tabel, tworzenie plików/tabel z przykładowymi danymi).

Metoda `TestDeCapitalize.tearDown(self)`

sprzątanie wykonywane po każdym teście (usuwanie tymczasowych plików etc).

Uruchomienie

```
python testy.py -v
```

```
testDziedzina (__main__.testy) ... ok
```

```
Nie zamieni poprawnych nazwisk ... ok
```

```
Proste sprawdzenia ... ok
```

```
Nie radzi sobie ... ok
```

```
-----
```

```
Ran 5 tests in 0.001s
```

```
OK
```

Zarządzanie zestawami testów

```
zestaw_testow.py
```

```
s1 = TestyDeCapitalize()  
s2 = modul.InneTesty()  
alltests = unittest.TestSuite([s1, s2])  
unittest.TextTestRunner(verbosity=3).run(alltests)
```

Testowanie za pomocą pakietu doctest

Przypomnienie

```
print modul.__doc__  
help(modul)
```

Testy w komentarzach

```
def deCapitalize(nazwisko):
```

```
    """
```

Zamiana napisów (nazwisk) pisanych wielkimi literami.

Przykłady:

```
>>> [deCapitalize(n) for n in ['Kaz WieLki', 'Stefan Batory']]  
['Kaz Wielki', 'Stefan Batory']
```

```
>>> deCapitalize('Henryk Walezy')
```

```
'Henryk Walezy'
```

```
>>> deCapitalize(2)
```

```
Traceback (most recent call last):
```

```
...
```

```
ArgumentNotStringError
```

```
"""
```

Uruchomienie testów

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

doctest — wynik

Trying:

```
[deCapitalize(n) for n in ['KazImieRz  
Wielki', 'Stefan Batory']]
```

Expecting:

```
['Kazimierz Wielki', 'Stefan Batory']
```

ok

Trying:

```
deCapitalize('Henryk Walezy')
```

Expecting:

```
'Henryk Walezy'
```

ok

Trying:

```
deCapitalize(2)
```

Expecting:

```
Traceback (most recent call last):
```

```
...
```

```
ArgumentNotStringError
```

ok

```
2 items had no tests:
  __main__
  __main__.ArgumentNotStringError
1 items passed all tests:
  3 tests in __main__.deCapitalize
3 tests in 3 items.
3 passed and 0 failed.
Test passed.
```

Pomiar wydajności fragmentu kodu

Klasa `timeit.Timer`

```
import timeit  
t = timeit.Timer(stmt='[6,5,4,3,2,1].sort()')  
print 'czas %.2f sec' % t.timeit()
```

Python 2.6

```
import timeit  
print 'czas %.2f sec' % timeit.timeit(stmt='[6,5,4,3,2,1].sort()')
```

Pomiar wydajności fragmentu kodu

Klasa `timeit.Timer`

```
import timeit  
t = timeit.Timer(stmt='[6,5,4,3,2,1].sort()')  
print 'czas %.2f sec' % t.timeit()
```

Python 2.6

```
import timeit  
print 'czas %.2f sec' % timeit.timeit(stmt='[6,5,4,3,2,1].sort()')
```

Pomiar wydajności całego programu

Z linii poleceń

```
$ python -m timeit '[3,2,1].sort()'
```

Wynik

1000000 loops, best of 3: 0.483 usec per loop

Profilowanie

Profilowanie dostarcza informacji o czasie wykonywania poszczególnych funkcji, liczbie wywołań etc.

Przykład profilowania

Wywołanie

```
$ python -m profile my_doctest.py
```

Wynik

```
19287 function calls (19035 primitive calls) in 0.350 CPU sec  
Ordered by: standard name  
ncalls tottime percall cumtime percall filename:lineno(function)  
1 0.000 0.000 0.010 0.010 :0(__import__)  
3 0.000 0.000 0.000 0.000 :0(_getframe)  
1 0.000 0.000 0.000 0.000 :0(allocate_lock)  
1436 0.000 0.000 0.000 0.000 :0(append)  
4 0.000 0.000 0.000 0.000 :0(callable)
```

pdb

Wywołanie

```
$ python -m pdb mymodule.py  
(pdb) help  
(pdb)
```