



UNIT testų kūrimas

Donatas Noreika

Kaip patikrinti, ar programa teisingai veikia?

Failas: keliamieji.py

```
def ar_keliamieji(metai):  
    if (metai % 400 == 0) or  
        (metai % 100 != 0 and metai % 4 == 0):  
        return ("Keliamieji")  
    else:  
        return ("Nekeliamieji")
```

```
print(ar_keliamieji(2000))  
print(ar_keliamieji(2020))  
print(ar_keliamieji(2100))
```

Keliamieji

Keliamieji

Nekeliamieji

Kaip ištestuoti programą UNIT testų pagalba

Failas: test_keliamieji.py

```
import unittest
from keliamieji import *

class TestKeliamieji(unittest.TestCase):

    def test_ar_keliamieji(self):
        rezultatas = ar_keliamieji(2000)
        lukestis = "Keliamieji"
        self.assertEqual(rezultatas, lukestis)
```

Testo paleidimas komandinėje eilutėje (cmd):

```
python -m unittest test_keliamieji.py
```

.

Ran 1 test in 0.000s

OK

Testo paleidimas tiesiogiai:

Faile test_keliamieji.py prirašyti:

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

Komandinėje eilutėje:

test_keliamieji.py

Pastaba: paleidžiant testą PyCharm programoje, to nereikia

Testo paleidimas PyCharm programoje:

Failas: test_keliamieji.py

```
import unittest
import keliamieji

class TestKeliamieji(unittest.TestCase):

    def test_ar_keliamieji(self):
        self.assertEqual(ar_keliamieji(2000), "Keliamieji")
        self.assertEqual(ar_keliamieji(2020), "Keliamieji")
        self.assertEqual(ar_keliamieji(2100), "Keliamieji")
```


Failas: test_keliamieji.py

```
import unittest
import keliamieji

class TestKeliamieji(unittest.TestCase):

    def test_ar_keliamieji(self):
        self.assertEqual(ar_keliamieji(2000), "Keliamieji")
        self.assertEqual(ar_keliamieji(2020), "Keliamieji")
        self.assertEqual(ar_keliamieji(2100),
            "Nekeliamieji")
```

```
Ran 1 test in 0.001s
OK
```

UNIT testų metodai:

Metodas	Tikrina	Python versija nuo
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Kai yra klaida kode:

Failas: keliamieji.py

```
def ar_keliamieji(metai):  
    if (metai % 400 == 0) or (metai % 4 == 0):  
        return ("Keliamieji")  
    else:  
        return ("Nekeliamieji")
```

Nekeliamieji != Keliamieji

Expected :Keliamieji

Actual :Nekeliamieji

Failas: aritmetika.py

```
def sudetis(a, b):  
    return a + b
```

```
def atimtis(a, b):  
    return a - b
```

```
def daugyba(a, b):  
    return a * b
```

```
def dalyba(a, b):  
    return a / b
```

Failas: test_aritmetika.py

```
import unittest  
import aritmetika
```

```
class TestAritmetika(unittest.TestCase):  
  
    def test_sudetis(self):  
        self.assertEqual(aritmetika.sudetis(10, 5), 15)  
        self.assertEqual(aritmetika.sudetis(-1, 1), 0)  
        self.assertEqual(aritmetika.sudetis(-1, -1), -2)
```

```
def test_atimtis(self):  
    self.assertEqual(aritmetika.atimtis(10, 5), 5)  
    self.assertEqual(aritmetika.atimtis(-1, 1), -2)  
    self.assertEqual(aritmetika.atimtis(-1, -1), 0)  
  
def test_daugyba(self):  
    self.assertEqual(aritmetika.daugyba(10, 5), 50)  
    self.assertEqual(aritmetika.daugyba(-1, 1), -1)  
    self.assertEqual(aritmetika.daugyba(-1, -1), 1)  
  
def test_dalyba(self):  
    self.assertEqual(aritmetika.dalyba(10, 5), 2)  
    self.assertEqual(aritmetika.dalyba(-1, 1), -1)  
    self.assertEqual(aritmetika.dalyba(-1, -1), 1)
```

Ran 4 tests in 0.001s

OK

Failas: aritmetika.py

```
def sudetis(a, b):  
    return a + b
```

```
def atimtis(a, b):  
    return a - b
```

```
def daugyba(a, b):  
    return a ** b
```

```
def dalyba(a, b):  
    return a / b
```


50 != 100000

Expected :100000

Actual :50

Failas: aritmetika.py

```
def sudetis(a, b):  
    return a + b
```

```
def atimtis(a, b):  
    return a - b
```

```
def daugyba(a, b):  
    return a * b
```

```
def dalysba(a, b):  
    return a // b
```

Ran 4 tests in 0.001s

OK

Faile test_aritmetika.py

```
def test_dalyba(self):  
    self.assertEqual(aritmetika.dalyba(10, 5), 2)  
    self.assertEqual(aritmetika.dalyba(-1, 1), -1)  
    self.assertEqual(aritmetika.dalyba(-1, -1), 1)  
    self.assertEqual(aritmetika.dalyba(5, 2), 2.5)
```

2.5 != 2

Expected :2

Actual :2.5

Faile aritmetika.py:

```
def dalyba(a, b):  
    if b == 0:  
        raise ValueError("Dalyba iš nulio negalima")  
    return a / b
```

Faile test_aritmetika.py:

```
class TestAritmetika(unittest.TestCase):  
  
    def test_dalyba(self):  
        self.assertEqual(aritmetika.dalyba(10, 5), 2)  
        self.assertEqual(aritmetika.dalyba(-1, 1), -1)  
        self.assertEqual(aritmetika.dalyba(-1, -1), 1)  
        self.assertEqual(aritmetika.dalyba(5, 2), 2.5)  
        self.assertRaises(ValueError, aritmetika.dalyba,  
10, 0)
```

ARBA:

```
def test_dalyba(self):  
    self.assertEqual(aritmetika.dalyba(10, 5), 2)  
    self.assertEqual(aritmetika.dalyba(-1, 1), -1)  
    self.assertEqual(aritmetika.dalyba(-1, -1), 1)  
    self.assertEqual(aritmetika.dalyba(5, 2), 2.5)  
    with self.assertRaises(ValueError):  
        aritmetika.dalyba(10, 0)
```

Failas: keliamieji2.py

```
def ar_keliamieji2(metai):  
    return (metai % 400 == 0) or (metai % 100 !=  
0 and metai % 4 == 0)
```


Failas: test_keliamieji2.py

```
import unittest
from keliamieji2 import *

class TestKeliamieji2(unittest.TestCase):

    def test_ar_keliamieji(self):
        result = ar_keliamieji2(2000)
        self.assertTrue(result)
        result = ar_keliamieji2(2020)
        self.assertTrue(result)
        result = ar_keliamieji2(2100)
        self.assertFalse(result)
```

Ran 1 test in 0.001s

OK

Objektų klasių testavimas

Failas: keliamieji3.py

```
class Keliamieji:  
  
    def tikrinti(self, metai):  
        return (metai % 400 == 0) or (metai %  
100 != 0 and metai % 4 == 0)
```

```
def diapazonas(self, nuo, iki):  
    sarasas = []  
    for metai in range(nuo, iki):  
        if (metai % 400 == 0) or (metai %  
100 != 0 and metai % 4 == 0):  
            sarasas.append(metai)  
    return sarasas
```

Failas: test_keliamieji3.py

```
import unittest
from keliamieji3 import *

class TestKeliamieji3(unittest.TestCase):

    def test_tikrinti(self):
        objektas = Keliamieji()
        self.assertTrue(objektas.tikrinti(2000))
        self.assertTrue(objektas.tikrinti(2020))
        self.assertFalse(objektas.tikrinti(2100))
```

```
def test_diapazonas(self):  
    objektas = Keliamieji()  
    rezultatas = objektas.diapazonas(1980, 2000)  
    lukestis = [1980, 1984, 1988, 1992, 1996]  
    self.assertEqual(rezultatas, lukestis)
```

Patogesnis būdas:

Failas: test_keliamieji3.py

```
import unittest
from keliamieji3 import *

class TestKeliamieji3(unittest.TestCase):
    def setUp(self):
        self.objektas = Keliamieji()
```

```
def test_tikrinti(self):  
    self.assertTrue(objektas.tikrinti(2000))  
    self.assertTrue(objektas.tikrinti(2020))  
    self.assertFalse(objektas.tikrinti(2100))  
  
def test_diapazonas(self):  
    rezultatas = objektas.diapazonas(1980, 2000)  
    lukestis = [1980, 1984, 1988, 1992, 1996]  
    self.assertEqual(rezultatas, lukestis)
```


UNIT testų privalumai

- Galimybė išvengti klaidų rašant ar taisant kodą
- UNIT testai gali būti panaudoti kaip būsimos programos dokumentacija
- Sutaupo laiko testuotojų komandai
- Taupo pinigus (klaidų taisymas vėliau yra brangus)

Testavimu paremtas programavimas (TDD)

Iš pradžių sukuriame testą – po to parašome kodą

Užduotis 1 (pagal 5-1)

- Sukurti UNIT testą 5 paskaitos 1 užduoties programai
- Kiekvienai funkcijai turi būti mažiausiai 3 patikrinimai
- Kai kurias funkcijas galima perdaryti taip, kad jos gražintų duomenis vietoje spausdinimo
- Maksimaliai patobulinti kodą, nuolatos leidžiant sukurtą UNIT testą

Užduotis 2 (pagal 6-1)

- Sukurti UNIT testą 6 paskaitos 1 užduoties programai
- Kiekvienai funkcijai turi būti mažiausiai 3 patikrinimai
- Maksimaliai patobulinti kodą, nuolatos leidžiant sukurtą UNIT testą

Užduotis 3

Nuosekliai, papunkčiui, pagal duotą UNIT testą sukurti programą, skaičiuojančią KMI:

```
import unittest
from uzduotis_14_3 import *

class TestUzduotis14_3(unittest.TestCase):
    def test_kmi(self):
        self.assertEqual(kmi(78, 1.82), 23.54788069073783)
        self.assertEqual(kmi(50, 1.56), 20.5456936226167)
        self.assertEqual(kmi(100, 1.90), 27.70083102493075)
        with self.assertRaises(ValueError):
            kmi(20, 1.40)
        with self.assertRaises(ValueError):
            kmi(240, 1.40)
        with self.assertRaises(ValueError):
            kmi(80, 0.40)
        with self.assertRaises(ValueError):
            kmi(80, 3.40)
```

pabaiga