

Programmeerimise süvendatud algkursus ITI0140

2014

Testimine

- Miks testida?
- Kuidas testida?
- Mida testida?
- Kui palju testida?

Miks testida?

- Koodi silumine (*debugging*) ja käsitsi testimine on hea, aga:
 - ei saa automatiseerida
 - tülikas: peab käsitsi erinevaid olukordi proovima
 - vajab arenduskekkonna tuge
 - iga muudatuse korral peab kõik uuesti läbi proovima
- Testimine aitab leida olukordi, mille puhul kood ei toimi korrektselt (tegeliku põhjuse leidmine on omaette teema)
- Testimine aitab tagada, et koodi parandamisel ei läheks rakendus kuskilt mujalt katki
- Korralikud testid täiendavad ülesande püstitust ja aitavad verifitseerida rakenduse vastavust spetsifikatsioonile
 - kord kirjutatud, saab teste käivitada korduvalt ja automaatselt

Kuidas testida?

- Ühiktestidega (*unit testing*)
 - kontrollime rakenduse toimimist võimalikult väikesel tasemel, st testime üksikut funktsiooni või klassi toimimist
- Integratsiooni (lõimumise) testidega (*integration testing*)
 - kontrollime rakenduse eri komponentide ja liideste omavahelist toimimist ja vastavust rakenduse disainile
- Süsteemi testimine
 - kontrollime tervikrakenduse vastavust nõuetele
- Lisaks veel kasutajaliidese testimine, kasutusmugavuse testimine, koormustestimine, jms.

Mida testida?

- Testida on vaja:
 - tüüpolukordasid
 - nendega kontrollime testitava osa üldist toimimist (näiteks $9/2 = 4.5$)
 - piirjuhte
 - nendega kontrollime, mis juhtub kriitilistes situatsioonides (näiteks $9/0=?$)
 - absurdseid olukordasid
 - nendega kontrollime sisendit, eriti kasutaja oma (näiteks $9/a=?$)
 - algoritmilist efektiivsust
 - kas on kasutatud efektiivset algoritmi
 - kas on kasutatud õigeid andmestruktuure (meenutage järjendeid ja hulki kodutöödest)
 - erinditöötlust
 - kontrollime, mis juhtub, kui tekivad erindid

Kui palju testida?

- Testida tuleb piisavalt palju ja hoolikalt
- Ühiktestimisel on hea vaadata näiteks testidega kaetud koodiridasid (*code coverage*)
 - sellega saame teada, et kõik kirjutatud kood töötab
 - samas ei saa teada, kui unustame mõnda olukorda testida
 - üldiselt loetakse juba päris heaks, kui 70-80% koodist on kaetud
- Testidega ei õnnestu kunagi kõiki olukordasid tuvastada ja tekitada
 - küll aga, mida olulisem on süsteem, seda rohkem ressursi kulub testimisele
 - sest testida on üldiselt odavam, kui likvideerida hiljem vigadest tulenevaid tagajärgi

Ühiktestimine Pythonis: ruutvõrrandi lahendamine

```
import math

def solve_quadratic_equation(a, b, c):
    """
    Solving quadratic equation  $ax^2 + bx + c = 0$ 

    Args:
    a --  $ax^2$ 
    b --  $bx$ 
    c --  $c$ 
    Returns:
    (x1, x2) tuple
    """
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    x2 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)

    return (x1, x2)
```

Test 1 toimib

```
import unittest

class Testing(unittest.TestCase):
    def test_1(self):
        self.assertEqual(solve_quadratic_equation(1, -3, 2), (1, 2))

if __name__ == "__main__":
    unittest.main(verbosity=5)
```

- Kontrollime, kas funktsioon üldse toimib
 - $x^2 - 3x + 2 = 0$
 - $x_1 = 1, x_2 = 2$

```
test_1 (__main__.Testing) ... ok
```

```
-----
Ran 1 test in 0.001s
```

```
OK
```


Test 2 toimib

```
import unittest

class Testing(unittest.TestCase):
    def test_2(self):
        self.assertEqual(solve_quadratic_equation(1, -4, 3.75), (1.5, 2.5))

if __name__ == "__main__":
    unittest.main(verbosity=5)
```

- Kontrollime, kas funktsioon üldse toimib
 - $x^2 - 4x + 3.75 = 0$
 - $x_1 = 1.5, x_2 = 2.5$

```
test_1 (__main__.Testing) ... ok
test_2 (__main__.Testing) ... ok
```

```
Ran 2 tests in 0.000s
```

```
OK
```

Test 3

```
import unittest

class Testing(unittest.TestCase):
    def test_3(self):
        self.assertEqual(solve_quadratic_equation(1, -4, 4), (2,))

if __name__ == "__main__":
    unittest.main(verbosity=5)
```

- Kontrollime, mis juhtub ühe lahendiga ruutvõrrandi korral (st ootame ennikut kahest elemendist, millest on ainult üks olemas)
 - $x^2 - 4x + 4 = 0$
 - $x_1 = 2$

Test 3 ei toimi

```
test_1 (__main__.Testing) ... ok
test_2 (__main__.Testing) ... ok
test_3 (__main__.Testing) ... FAIL
```

```
=====
FAIL: test_3 (__main__.Testing)
```

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

```
  File "...", line 28, in test_3
```

```
    self.assertEqual(solve_quadratic_equation(1, -4, 4), (2,))
```

```
AssertionError: Tuples differ: (2.0, 2.0) != (2,)
```

```
First tuple contains 1 additional elements.
```

```
First extra element 1:
```

```
2.0
```

```
- (2.0, 2.0)
```

```
+ (2,)
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (failures=1)
```

Täiendame tagastatavat väärtust

```
import math

def solve_quadratic_equation(a, b, c):
    """
    Solving quadratic equation  $ax^2 + bx + c = 0$ 

    Args:
    a --  $ax^2$ 
    b --  $bx$ 
    c --  $c$ 
    Returns:
    (x1, x2) tuple
    """
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    x2 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)

    return (x1, x2) if x1 != x2 else (x1,)
```

Test 3 toimib

```
test_1 (__main__.Testing) ... ok  
test_2 (__main__.Testing) ... ok  
test_3 (__main__.Testing) ... ok
```

```
Ran 3 tests in 0.000s
```

```
OK
```

Test 4

```
import unittest

class Testing(unittest.TestCase):
    def test_4(self):
        self.assertEqual(solve_quadratic_equation(1, 1, 1), None)

if __name__ == "__main__":
    unittest.main(verbosity=5)
```

- Kontrollime, mis juhtub puuduva lahendi korral, st kontrollime juurealust
 - $x^2 + x + 1 = 0$

Test 4 ei toimi

```
test_1 (__main__.Testing) ... ok
test_2 (__main__.Testing) ... ok
test_3 (__main__.Testing) ... ok
test_4 (__main__.Testing) ... ERROR
```

```
=====  
ERROR: test_4 (__main__.Testing)
```

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

```
  File "...", line 30, in test_4
```

```
    self.assertEqual(solve_quadratic_equation(1, 1, 1), None)
```

```
  File "...", line 14, in solve_quadratic_equation
```

```
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
```

```
ValueError: math domain error
```

```
-----  
Ran 4 tests in 0.001s
```

```
FAILED (errors=1)
```

Lisame juurealuse avaldise kontrolli

```
def solve_quadratic_equation(a, b, c):  
    """  
    Solving quadratic equation  $ax^2 + bx + c = 0$   
  
    Args:  
    a --  $ax^2$   
    b --  $bx$   
    c --  $c$   
    Returns:  
    (x1, x2) tuple  
    """  
    if b**2 - 4 * a * c < 0:  
        return None  
  
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)  
    x2 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)  
  
    return (x1, x2) if x1 != x2 else (x1,)
```


Test 4 toimib

```
test_1 (__main__.Testing) ... ok  
test_2 (__main__.Testing) ... ok  
test_3 (__main__.Testing) ... ok  
test_4 (__main__.Testing) ... ok
```

```
Ran 4 tests in 0.001s
```

```
OK
```

Test 5

```
import unittest

class Testing(unittest.TestCase):
    def test_5(self):
        self.assertEqual(solve_quadratic_equation(0, 1, 1), None)

if __name__ == "__main__":
    unittest.main(verbosity=5)
```

- Kontrollime, mis juhtub puuduva lahendi korral, st kontrollime nulliga jagamist
 - $0x^2 + x + 1 = 0$

Test 5 ei toimi

```
test_1 (__main__.Testing) ... ok
test_2 (__main__.Testing) ... ok
test_3 (__main__.Testing) ... ok
test_4 (__main__.Testing) ... ok
test_5 (__main__.Testing) ... ERROR
```

```
=====
ERROR: test_5 (__main__.Testing)
```

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

```
  File "...", line 35, in test_5
```

```
    self.assertEqual(solve_quadratic_equation(0, 1, 1), None)
```

```
  File "...", line 17, in solve_quadratic_equation
```

```
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
```

```
ZeroDivisionError: float division by zero
```

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

```
FAILED (errors=1)
```

Lisame nulliga jagamise kontrolli

```
def solve_quadratic_equation(a, b, c):  
    """  
    Solving quadratic equation  $ax^2 + bx + c = 0$   
  
    Args:  
    a --  $ax^2$   
    b --  $bx$   
    c --  $c$   
    Returns:  
    (x1, x2) tuple  
    """  
    if b**2 - 4 * a * c < 0 or a == 0:  
        return None  
  
    x1 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)  
    x2 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)  
  
    return (x1, x2) if x1 != x2 else (x1,)
```

Test 5 toimib

```
test_1 (__main__.Testing) ... ok  
test_2 (__main__.Testing) ... ok  
test_3 (__main__.Testing) ... ok  
test_4 (__main__.Testing) ... ok  
test_5 (__main__.Testing) ... ok
```

```
Ran 5 tests in 0.000s
```

```
OK
```

Robotite ülesande kohta

Tähtaeg on **järgmine** tund (teisipäeval, st järgmine tund esitades saab ikka täispunktid)

- Aine kodulehel on täiendatud nii simulaatorit (praegu ver 4) kui loenguslaide.
- Tasub alati vaadata kodulehte enne kui koodimissessiooni alustate - ehk on midagi uuendatud või täiendatud!

Ülesanne: Rooma numbrid

Teid võetakse arendajana tööle vingesse idufirmasse. Peale kiiret sisseelamist selgub, et enne teid töötas samal ametipostil isik, kes ei ole kindlasti käesolevat õppeainet läbinud. Nimelt, teile antakse ülesanne eelnevalt mainitud programmeerija koodi testida ja vajadusel siluda – koodile peale vaadates ohkate ja asute tööle.

Ülesanne: Rooma numbrid

- Lae alla aine kodulehelt "**roman_numerals.py**"
- Ülesandeks on **kirjutada testid**, mis kinnitavad, et teisendusalgorithm toimib või leida sellised sisendid, mille puhul ei toimi.
- Vea leides peate antud juhul koodis vigase koha ära parandama (võite ilusamaks ja loogilisemaks koodi teha).

Analüüsida antud koodi ja kirjutada kõik vajalikud testid käsitsi. Üldiseks eesmärgiks on katta testidega kõik koodi read ja kõik võimalused. Kuidas olla kindel, et kõik vead said leitud (isegi kui kõik koodiread on kaetud)?