Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike - smjer matematika i računarstvo

Renato Dean

Automatizirano testiranje

Seminarski rad

Osijek, 2019.

Sadržaj

Poglavlje 1

Uvod

Svaki programer je barem jednom u životu testirao svoj program, svjesno ili nesvjesno, eksperimentirajući sa raznim ulaznim podacima za koje zna što program treba vratiti. Takvo testiranje se naziva istraživačko i jedan je od oblika ručnog testiranja. Kako bi softver bio u cijelosti testiran, trebali bismo proći kroz sve njegove funkcionalnosti i testirati vraća li dobar rezultat. Svaki puta kada napravimo nekakvu promjenu, ukoliko testiramo ručno, moramo pokriti sve nove funkcionalnosti i provjeriti jesu li izmijenjene neke od starih te iznova prilagoditi testove. Jer je taj posao monoton te jer je čovjek sklon pravljenju pogrešaka, zadatak testiranja možemo prepustiti nekoj skripti ili aplikaciji. U ovom radu ćemo se bazirati na testiranju programa u Python programskom jeziku.

Poglavlje 2

Testiranja u Pythonu

2.1 Unit test

2.1.1 Jednostavno testiranje

Ideja unit testiranja je provjera ispravnosti manje komponente u programu, kao što je npr. provjera ispravnosti neke funkcije. Za dani input, znamo što funkcija treba vratiti. Niže je prikazan primjer testiranja Pythonove funkcije sum() koja sumira brojeve u nekom iterabilnom objektu. Program treba ispisati navedenu poruku samo u slučaju da test **nije** uspio!

```
>>> assert sum([1, 2, 3]) == 6, "Should be 6"
>>>
```

Slika 2.1: Unit test je prošao

```
>>> assert sum([1, 2, 2]) == 6 , "Should be 6"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Should be 6
>>>
```

Slika 2.2: Unit test nije prošao

Prilikom pisanja ozbiljnijih testova ćemo sve testove stavljati u posebne datoteke pa čak i mape. Na sljedećoj slici vidimo testiranje funkcije sum u posebnoj datoteci.

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum2():
    assert sum((1, 2, 2)) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    test_sum2()
    print("Everything passed")
```

Slika 2.3: sumiranje.py

```
dean@Dean-Latitude-D630 ~/Radna površina/si_seminar $ python3 sumiranje.py
Traceback (most recent call last):
  File "sumiranje.py", line 9, in <module>
    test_sum2()
  File "sumiranje.py", line 5, in test_sum2
    assert sum((1, 2, 2)) == 6, "Should be 6"
AssertionError: Should be 6
```

Slika 2.4: output sumiranja.py

U izvještaju možemo primijetiti da funkcija test_sum_tuple() nije prošla test jer suma brojeva u toj uređenoj trojci nije 6.
Ukoliko više testova padne, prikladnije je koristiti složenije aplikacije koje imaju mnoštvo alata koje pomažu pri testiranju (npr. debugger), Takve aplikacije se zovu Test Runneri.

2.1.2 Test Runner

Najpoznatiji Pythonov Test Runner je unittest koji usput sadrži i framework za testiranje. On zahtijeva da su testovi metode u nekoj klasi te koristi specijalne assert metode iz klase unittest.TestCase. Pogledajmo kako se testira prethodni primjer koristeći unittest:

```
import unittest

class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum2(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

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

Slika 2.5: sumiranje_unittest.py

```
dean@Dean-Latitude-D630 ~/Radna površina/si_seminar $ python3 sumiranje_unittest
.py
.F
======================================================================
FAIL: test_sum2 (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "sumiranje_unittest.py", line 9, in test_sum2
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: 5 != 6 : Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Slika 2.6: unittest output

Još bolja opcija je koristiti pytest. On će automatski pokrenuti testove u datoteci koja ima prefiks 'test'. Sadrži još mogućnosti, kao što su npr. mogućnost ponovnog pokretanja na neuspješnom testu te se može proširiti sa mnoštvom pluginova. Pogledajmo primjer testiranja koristeći pytest:

```
def test_sum():
    assert sum([1, 2, 2]) == 6
```

Slika 2.7: test_sumiranje.py

```
================================= FAILURES =================================
_____________________________ test_sum _____________________________

    def test_sum():
>       assert sum([1, 2, 2]) == 6
E       assert 5 == 6
E        +  where 5 = sum([1, 2, 2])

test_sumiranje.py:2: AssertionError
========================== 1 failed in 0.06 seconds ==========================
```

Slika 2.8: pytest output

2.2 Integration test

Integracijski test provjerava ispravnost međusobne komunikacije komponenti u programu, kao što je npr. kontaktiranje API-ja ili komponiranje implementiranih funkcija. Testiranje se vrši slično kao i kod unit testa, uz razliku da se u testu vrši komunikacija između više komponenti. Zbog toga je povećana mogućnost "side-effecta", tj. mijenjanja stanja neke druge komponente. Generalno želimo razbiti komade koda na što manje cjeline kako bismo ih lakše testirali. Metode koje imaju puno side efekata zahtijevaju refactoring. Zbog toga je dobra praksa razdvojiti unit testove i integracijske testove u posebne foldere te organizirati projekt na sljedeći način:

```
project/
│
├── my_app/
│   └── __init__.py
│
└── tests/
    │
    ├── unit/
    │   ├── __init__.py
    │   └── test_sum.py
    │
    └── integration/
        ├── __init__.py
        └── test_integration.py
```

Slika 2.9: Primjer organizacije projekta

Niže je naveden primjer u kojima se testira komponiranje Pythonove gotove funkcije sumiranja sa korisnički implementiranom funkcijom kvadriranja

```
def kvadriraj(x):
    return x*x

def test_kvadriraj_sumu():
    assert kvadriraj(sum([1,2,2])) == 25
```

Slika 2.10: test_integration.py

```
dean@Dean-Latitude-D630 ~/Radna površina/si_seminar/integration $ pytest
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-4.5.0, py-1.8.0, pluggy-0.11.0
rootdir: /home/dean/Radna površina/si_seminar/integration
collected 1 item

test_integracije.py .                                                    [100%

========================== 1 passed in 0.10 seconds ===========================
```

Slika 2.11: Output pytesta

2.3 Automatizirano testiranje

2.3.1 Selenium

Selenium je alat namijenjen za automatizirano testiranje web softvera. Skripte za testiranje koje koriste Selenium mogu biti napisane u raznim programskim jezicima kao što su Java, Python, C, PhP, Ruby, . . . te se skripte mogu vrtiti na bilo kojem operacijskom sustavu i na bilo kojem browseru. Na sljedećim slikama možemo vidjeti primjer korištenja Selenium Webdrivera, Python skripta pokušava otvoriti Firefox, pristupiti Googleu i unijeti "Automated Testing" u njegov search bar te će ispisati poruku o uspješnosti.

```python
from selenium import webdriver

import unittest
class Sample(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.driver = webdriver.Firefox()
        cls.driver.implicitly_wait(10)
        cls.driver.maximize_window()
    def test_search_automatedtesting(self):
        self.driver.get("https://google.com")
        self.driver.find_element_by_name("q").send_keys("Automated Testing")

    @classmethod
    def tearDownClass(cls):
        cls.driver.close()
        cls.driver.quit()
        print("test completed")
```

Slika 2.12: Sampletest.py

Možemo primijetiti i jedan od nedostataka ovog open-source softvera: testni izvještaji nisu predetaljni. Uz to, testiranje slika nije moguće. Srećom, te stvari se mogu popraviti integriranjem s drugim softverima kao što je Sikuli za testiranje slika te JUnit za pisanje izvještaja.

```
dean@Dean-Latitude-D630 ~/Radna površina/si_seminar/selenium $ python3 -m unitte
st SampleTest.py
.test completed

----------------------------------------------------------------------
Ran 1 test in 12.245s

OK
```

Slika 2.13: Output unittesta Seleniuma

2.3.2 Travis CI

Dosadašnji primjeri su koristili ručno testiranje. Postoje alati koji automatski vrše testove kada napravimo promjene, kao što je npr. pushanje na Git. Ti alati su poznati kao CI/CD, a skraćenica dolazi od Continuous Integration/Continuous Deployment. U kombinaciji s Pythonom se često koristi Travis CI, koji je besplatan za open-source projekte. Samo se treba ulogirati na web-stranicu sa svojim GitHub ili GitLab podacima, napraviti datoteku naziva .travis.yml koja ima sljedeću strukturu:

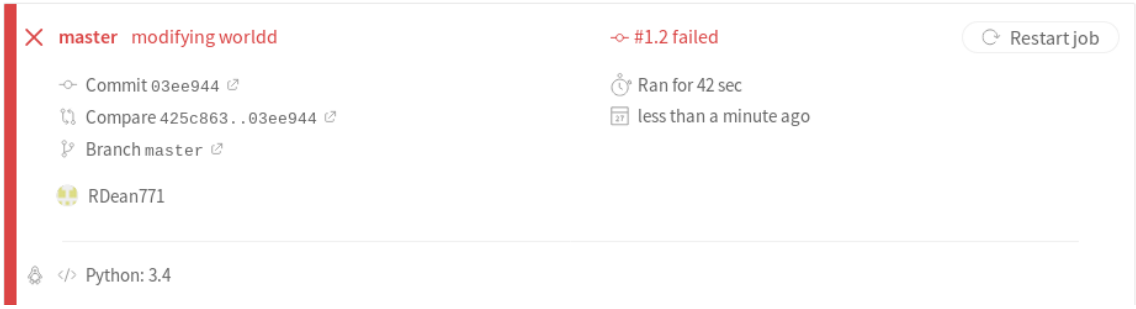
```yaml
language: python
python:
  - "2.7"
  - "3.7"
install:
  - pip install -r requirements.txt
script:
  - python -m unittest discover
```

Slika 2.14: Travis CI

Ovakva datoteka upućuje Travisa na to da:

- Testira za Python 2.7 i 3.7
- Instalira sve pakete izlistane u requirements.txt
- Pokrene testove koristeći unittest

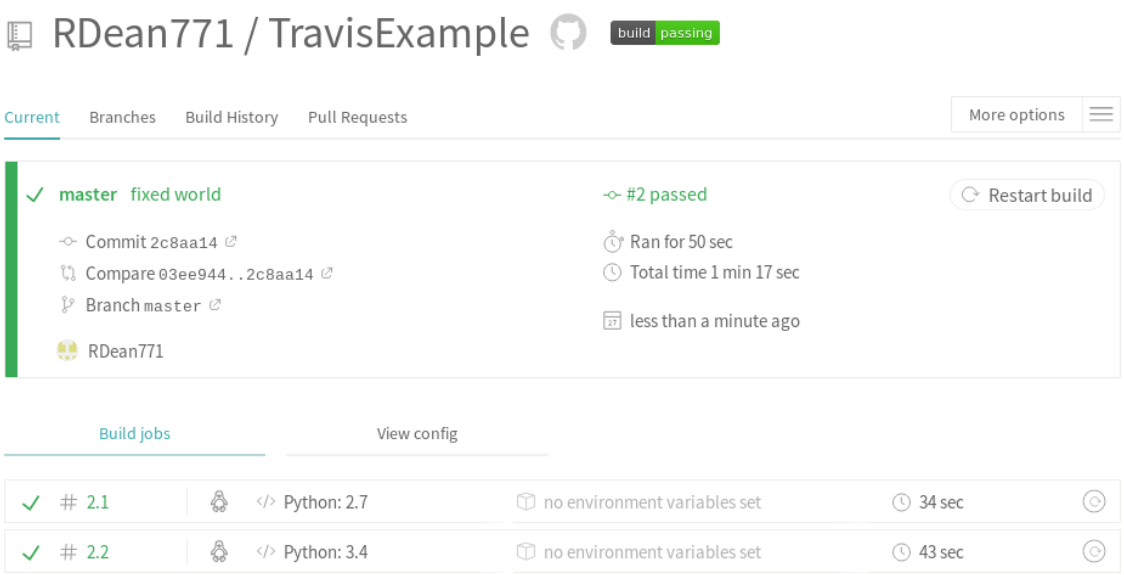
Nakon commitanja i pushanja toga na Git, svaki idući update repozitorija automatski pokreće Travisa te rezultate testa možemo vidjeti na njihovoj web stranici. Na sljedećim slikama je napisan trivijalan program koji treba ispisati ”Hello, world!”. Za njega je napisan test koji to provjerava. Prvo je na GitHub pushana verzija koja ispisuje ”Hello, worldd!” te je nakon toga pushana druga, popravljena verzija. Pogledajmo outpute na Travis web stranici te dijelove programskog koda:



Slika 2.15: Travis CI je pronašao error u buildu

```
448 platform linux -- Python 3.4.6, pytest-3.0.7, py-1.4.33, pluggy-0.4.0 -- /home/travis/virtualenv/python3.4.6/bin/python
449 cachedir: .cache
450 rootdir: /home/travis/build/RDean771/TravisExample, inifile:
451 collected 1 items
452
453 tests/test_hello.py::test_says_world FAILED
454
455 ================================== FAILURES ==================================
456 _____________________________ test_says_world ______________________________
457
458     def test_says_world():
459 >       assert hello.say_what() == 'worldd'
460 E       AssertionError: assert 'world' == 'worldd'
461 E         - world
462 E         + worldd
463 E         ?      +
464
465 tests/test_hello.py:5: AssertionError
466 ========================== 1 failed in 0.03 seconds ==========================
467 The command "py.test -v" exited with 1.
468
469
470
471 Done. Your build exited with 1.
```

Slika 2.16: Prikaz pogreške



Slika 2.17: Prikaz uspješnog testiranja

```
from __future__ import print_function
import sys


def hello(what):
    print('Hello, {}!'.format(what))


def say_what():
    return 'world'


def main():
    hello(say_what())
    return 0


if __name__ == '__main__':
    sys.exit(main())
```

Slika 2.18: hello.py

```
  GNU nano 2.5.3          Datoteka: .travis.yml

language: python
sudo: false
python:
- 2.7
- 3.4
script: py.test -v
```

Slika 2.19: .travis.yml

```
  GNU nano 2.5.3          Datoteka: test_hello.py

import hello


def test_says_world():
    assert hello.say_what() == 'world'
```

Slika 2.20: test_hello.py

2.4 Zaključak

Automatizirano testiranje se smatra ključnim u velikim firmama jer je računalo znatno bolje u monotonim, ponavljajućim poslovima nego čovjek. Isto tako, ubrzava proces testiranja i detekcije grešaka u softveru. Radno vrijeme računala može biti 24/7, dok čovjekovo ne može. Testiranje koda se može paralelno vrtiti na jako puno uređaja, dok bi ručna provjera trajala jako dugo, ukoliko bi se uopće i mogla izvesti. Zbog svega toga, dobit ćemo kvalitetniji softver ranije, uz korištenja manje resursa. Ipak, treba biti oprezan. Ti testovi neće sami po sebi popravljati greške te bi se trebali koristiti za detekciju bugova prilikom izvođenja jednostavnih operacija, kao što je npr. slanje maila kada netko zaboravi šifru prilikom logiranja na svoj account.