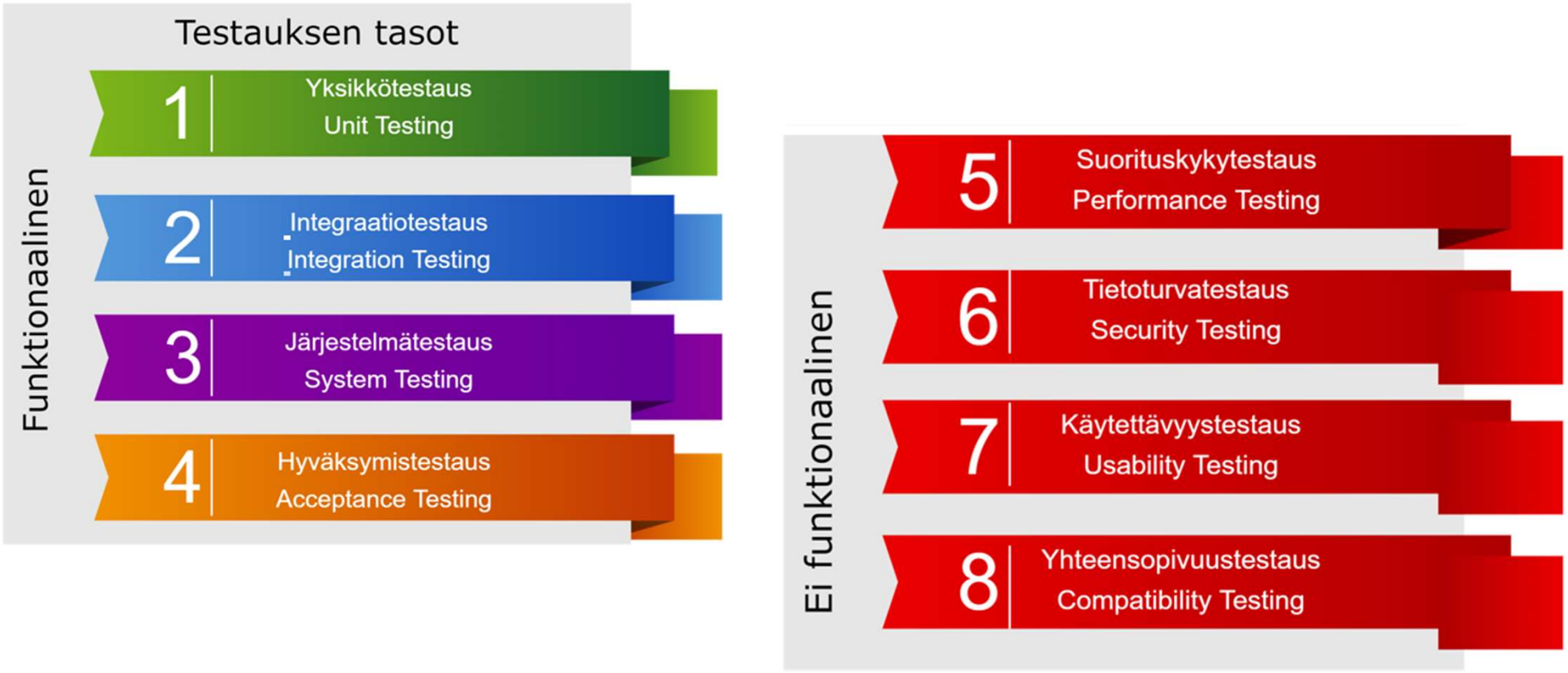
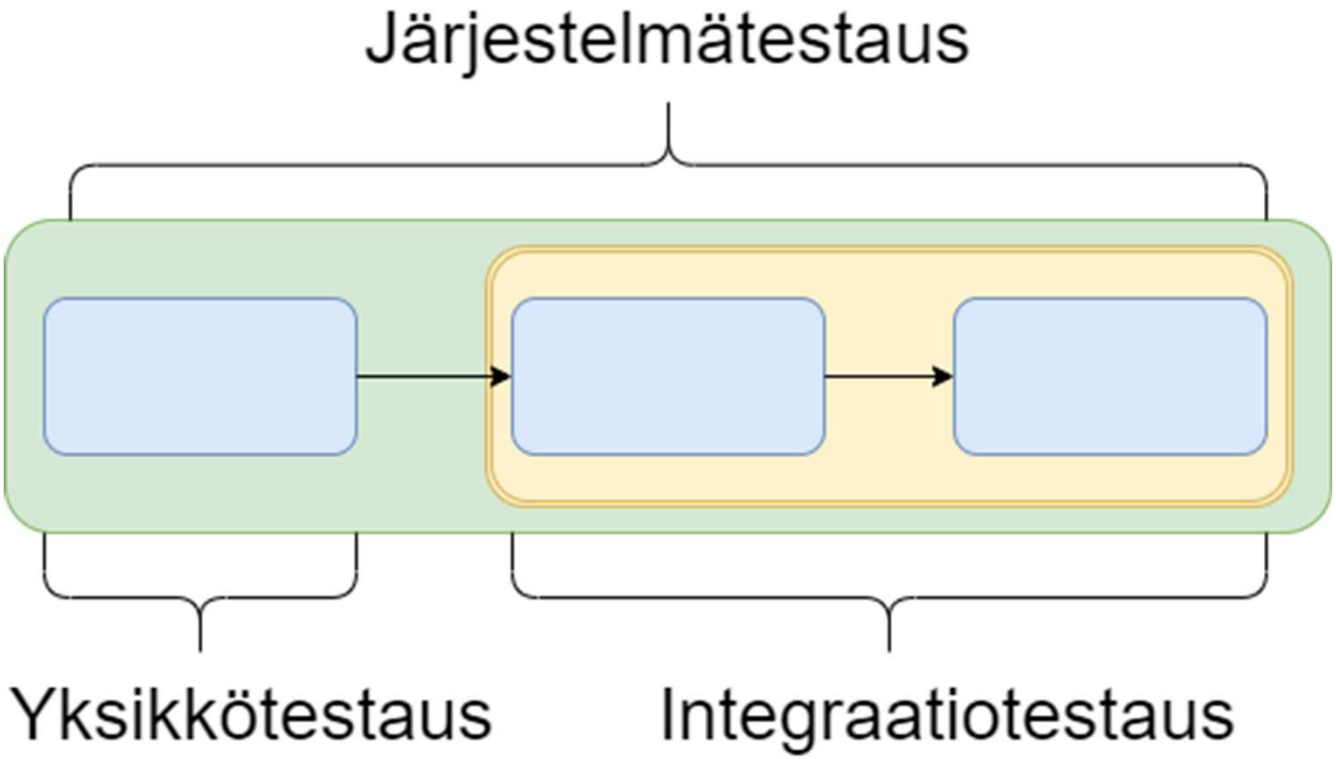
AW Academy

Python Testaus

Testaaminen



Testaaminen



Yksikkötestaus

- Yksikkötestaus on ohjelmatestaustauksen osa-alue.
- Yksikkötesteillä testataan sovelluksen yksittäisiä komponentteja (esim luokka/funktio/metodi).
- Yksikkötestauksen tavoitteena on varmistaa testien avulla, että ohjelma toimii kuten sen halutaan toimivan joka tilanteessa.
- Kattavat testit ennaltaehkäisevät bugien syntymistä.
- Yksikkötestauksen suorittaa yleensä ohjelmoija.
  - Isoissa yrityksissä saattaa olla oma QA (Quality Assurance) tiimi, joka hoitaa testauksen
- Python kielellä yksikkötestaamiseen käytetään esimerkiksi unittest viitekehystä (framework)
  - Dokumentaatio https://docs.python.org/3/library/unittest.html#

Unit test esimerkki

```
import unittest
import testaus.summa as summa
# from .. import summa


class TestSumma(unittest.TestCase):

    def test_sum_numbers_returns_sum_of_integers(self):
        result = summa.sum_numbers(2, 3)
        self.assertEqual(result, 5)

    def test_sum_numbers_works_with_negative_numbers(self):
        result = summa.sum_numbers(-2, -3)
        self.assertEqual(result, -5)

    def test_sum_numbers_returns_sum_of_floats(self):
        result = summa.sum_numbers(2.4, 3.2)
        self.assertEqual(result, 5.6)

    def test_sum_numbers_returns_sum_of_integer_and_float(self):
        result = summa.sum_numbers(2, 3.2)
        self.assertEqual(result, 5.2)

    def test_sum_numbers_takes_rises_error_if_parameters_are_not_numbers(self):
        with self.assertRaises(TypeError):
            summa.sum_numbers(1, "a")


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

- Testausta suoritetaan antamalla sovellukselle joukko syötteitä ja varmistamalla, että saadut vastauksen ovat halutun kaltaisia.
  - Tapahtuu yleensä väittämien (assertio) avulla
  - Väittämä on lause, joka on arvoltaan joko tosi tai epätosi (testi joko menee läpi tai ei)

Arrage - Act – Assert (AAA)

```
from unittest import TestCase
import testaus.calculator.tulo as tulo
import unittest

class TestTulo(TestCase):

    def test_multiply_numbers(self):
        # Arrange
        num1 = num2 = 2
        expected_result = 4
        # Act
        result = tulo.multiply_numbers(num1, num2)
        # Assert
        self.assertEqual(result, expected_result)


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

Yksi suosittu tapa muotoilla testi on käyttää AAA formaattia

Arrange kohdassa esitellään muuttujat

Act kohdassa tehdään toiminto, esim funktiokutsu

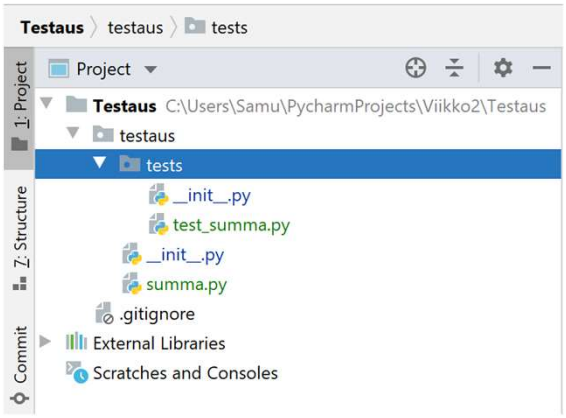
Assert kohdassa varmistetaan väittämien avulla, että tulos oli halutun kaltainen

Näin testaat parhaiden käytänteiden mukaan

- **Testaa automaattisesti**
  - Suoritetaan testit aina kun koodiin tehdään muutoksia CI/CD
- **Testaa kattavasti**
  - Testataan kaikki mikä sovelluksessa voi mennä Rikki
- **Testaa riippumattomasti**
  - Testien ei tule olla riippuvaisia muista testeistä, eikä ympäristöstä
- **Testaa laadukkaasti**
  - Testit tulee tehdä huolella, sillä ne ovat yhtä tärkeitä, kuin itse tuotantokoodi

Hyvä tietää

- Jokaista testattavaa luokkaa/tiedostoa varten kirjoitetaan erillinen testi tiedosto/luokka
  - Esim henkilo.py tiedoston testaamiseen luodaan test_henkilo.py
- **Testi metodien nimestä tulee löytyä test**, jotta ne toimivat testeinä
- Python testit voidaan ajaakomentoriviltä komennolla **python -m unittest**
  - Alla esimerkki projektin rakenteesta ja testien suorittamisesta komentoriviltä




```
(base) C:\Users\Samu\PycharmProjects\Viikko2\Testaus>python -m unittest
.....
----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

TDD (Test Driven Development)

- TDD on sovelluskehitystekniikka (ei testaustekniikka).
- TDD mukaan testit kirjoitetaan aina ennen toiminnallisuutta
- TDD pakottaa suunnittelemaan sovellusta testien kautta, ja näin saadaan ennaltaehkäistyä mahdollisia bugeja.
- Yksi hyvä tapa on kirjoittaa lista testeistä, jotka täytyy toteuttaa, ja valita aina uusi testi tehtäväksi kun edellinen ja siihen liittyvä ohjelmalogiikka ovat valmiita.
- Älä yritä ennakoida pitkälle vaan tee vain välttämättömät muutokset, jotta koodi toimii kuten testissä on määritelty

TDD (Test Driven Development)

Harjoitukset

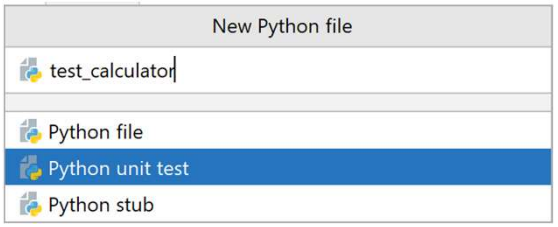
- Harjoitus 1: Laskimen testaus
- Harjoitus 2: TDD Lauseen muokkaaja
- Harjoitus 3: TDD FizzBuzz
- Harjoitus 4: TDD Tiedoston lukija
- Harjoitus 5 - lisätehtävä: Mock - tiedostopolku
- Harjoitus 6 - lisätehtävä: Mock - hakemistolistaus

Harjoitus 1: Laskimen testaus 1/2

1. Aloita uusi Python projekti.
2. Luo projektiin seuraavanlainen hakemistorakenne

```
Viikko2/
        testaus/
                __init__.py
                tests/
                        __init__.py
                calculator/
                        __init__.py
                        calculator.py
```

3. Tee calculator.py tiedostoon funktio “plus”, jonka avulla voi laskea yhteen kaksi numeroa, jotka tulevat funktiolle parametrinä
4. Tee seuraavaksi tests pakettiin test_calculator niminen python unit test tiedosto
   - Tämä tekee valmiiksi rungon testiluokalle.



- ---> Jatkuu

Harjoitus 1: Laskimen testaus 2/2

5. Anna luokalle uusi nimi, esim TestClaculator
6. Voit pyyhkiä valmiin metodin pois tai käyttää sitä pohjana uudelle testille.
7. Mieti seuraavaksi miten testaisit plus function toimintaa ja kirjoita sille sitten testit
   - Mitä pitää palauttaa? Ovathan parametrit varmasti numeroita? …
8. Kirjoita seuraavaksi lisää funktioita laskimeesi (esim minus, jako ja kertolasku) tee näille myös testit
   - Tee yksi funktio kerrallaan ja sille testit

Harjoitus 2: TDD Lauseen muokkaaja

1. Vaatimukset
   Esimerkki
   - Funktio jolle syötetään lause parametrina
   - Kaikki sanat, joiden pituus on vähintään 5 tulee kääntää (Kissa -> assiK)
   - Kaikki sanat, joiden pituus on 2 tulee kirjoittaa isolla
2. Kirjoita testi ensin, sitten toiminnallisuutta. Refaktoroi tarvittaessa ja toista.
3. Lisätehtävä
   - Lauseen ensimmäinen kirjain tulee olla aina iso
   - Viimeinen merkki täytyy olla jokin seuraavista ”.!?” ja jos ei ole, niin loppuun lisätään ”.”

```
print(reverse_words("Pluto is a dog"))

otulP IS a dog
```

Harjoitus 3: TDD FizzBuzz

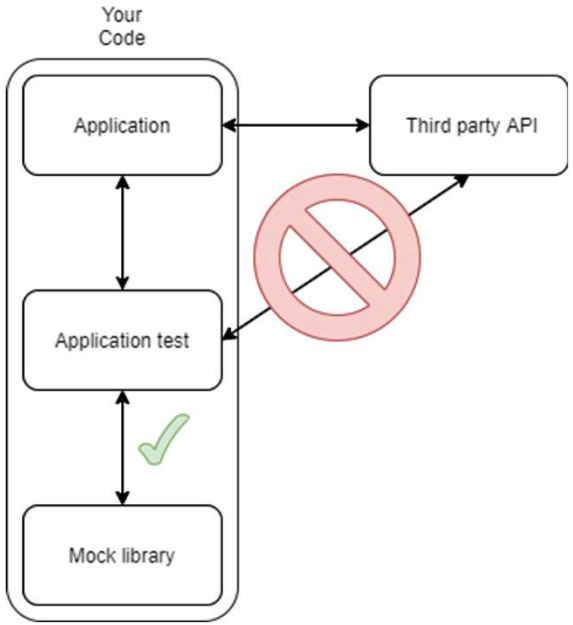
1. Vaatimukset
   - Ohjelma ottaa parametrina positiivisen kokonaisluvun
   - Kaikkien 3 jaollisten lukujen syötteellä palautetaan ”Fizz”
   - Kaikkien 5 jaollisten lukujen syötteellä palautetaan ”Buzz”
   - Kaikkien 3 ja 5 jaollisten lukujen syötteellä palautetaan ”FizzBuzz”
   - Muuten ohjelma palauttaa annetun kokonaisluvun
2. Kirjoita testi ensin, sitten toiminnallisuutta. Refaktoroi tarvittaessa ja toista.

Harjoitus 4: TDD Tiedoston lukija

1. Vaatimukset
   - Funktio, jolle syötetään tiedoston nimi, sekä rivinumero parametreinä.
   - Funktion tulee avata tiedosto ja palauttaa sieltä rivinumeron osoittama rivi merkkkijonona.
   - Jos tiedostoa ei löydy, funktion tulee palauttaa viesti ”File not found”.
   - Jos tiedosto on tyhjä, funktion tulee palauttaa ”Empty file”.
   - Jos tiedostossa on liian vähän rivejä, funktion tulee nostaa virhe ”IndexError”.
2. Kirjoita testi ensin, sitten toiminnallisuutta. Refaktoroi tarvittaessa ja toista.

Mock testaus

- Mock olioiden avulla voidaan simuloida testissä tarvittavien ulkoisten ohjelman osien käyttäytymistä.
  - Kaikki kirjastot, joiden koodiin ei päästä käsiksi (Olivatpa ne sitten omassa tai jonkun toisen osapuolen sovelluksessa).
  - Esim
    - Omassa sovelluksessa olevat toteuttamattomat toiminnallisuudet
    - Oman järjestelmän toinen mikropalvelu
    - Ulkoiset palvelut, kuten AWS palvelut
- Mockin avulla voidaan testata esim funktiota, joka hakee kuvan AWS s3 bucketista kuitenkaan kutsumatta s3 APIa testi koodissa.



Mock testaus – Mitä mockilla testataan?

- Mockilla voi testata esimerkiksi seuraavia asioita
  - Kutsuttiinko mock funktiota vai ei
  - Kuinka monta kertaa mock funktiota kutsuttiin
  - Mitä parametrejä annettiin, kun mock funktiota kutsuttiin

Mock testaus

- Mock oliot ovat ”vale olioita”, jotka simuloivat ulkoisen palvelun toimintaa.
- Mock oliolle kerrotaan, mitä sen tulisi missäkin tilanteessa palauttaa ja tällä tavalla sovelluksen koodin toimintaa saadaan testattua.
- mock.patch avulla saadaan kerrottua, mitä funktiota tai luokkaa mock olion tulisi simuloida

Esimerkki testi

```
from unittest import TestCase, mock
from src.words.mock_demo import get_file_path, get_env

class MockDemoTest(TestCase):
    def test_get_env_works_properly(self):
        with mock.patch('src.words.mock_demo.os') as mock_os:
            get_env('TEST_ENV')
            assert mock_os.getenv.call_count == 1

    def test_do_get_env_return_value(self):
        with mock.patch('src.words.mock_demo.os.getenv', return_value='test_value'):
            assert get_env('TEST_ENV') == 'test_value'
```

Testattava funktio

```
import os

def get_env(value):
    env = os.getenv(value)
    print(f'The environment variable value is {env}')
    return env
```

Harjoitus 5 - lisätehtävä: Mock - tiedostopolku

1. Toteuta funktio, joka palauttaa python tiedostosi tiedostopolun.
2. Kirjoita tälle vastaavat testit , kuin aiemmassa esimerkissä.

Harjoitus 6 - lisätehtävä: Mock - hakemistolistaus

1. Toteuta funktio, joka palauttaa listan tiedostoista, jotka löytyvät funktiolle parametrinä annettavasta tiedostopolusta.
2. Kirjoita tälle vastaavat testit , kuin aiemmassa esimerkissä.

Test Doubles

- Vaikka yksikkötesteillä ja Mockeilla pääsee jo pitkälle, niin välillä tarvitaan muitakin testaus menetelmiä
- Ulkoisten palveluiden testaamiseen käytetään mockien lisäksi usein myös **Stub** ja **Fake** testejä
  - Näistä ja mockeista lisäinfoa esim täältä https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

Linkkejä, vinkkejä ja lisämateriaalia aiheesta

1. Lisätietoa testaamisesta (koko sivusto)
   1. https://www.softwaretestinghelp.com/
2. Yksikkötestaaminen
   1. https://docs.python.org/3/library/unittest.html
   2. http://www.cse.hut.fi/fi/opinnot/CSE-A1121/2015/softwaretesting/pythontesting.html
3. Mock testaaminen
   1. https://medium.com/@yeraydiazdiaz/what-the-mock-cheatsheet-mocking-in-python-6a71db997832
   2. https://docs.python.org/3/library/unittest.mock-examples.html
   3. https://realpython.com/testing-third-party-apis-with-mocks/#your-first-mock
   4. https://realpython.com/python-mock-library/
4. TDD
   1. https://www.freecodecamp.org/news/learning-to-test-with-python-997ace2d8abe/
5. 180+ Testitapausta web sovellukselle
   1. https://www.softwaretestinghelp.com/sample-test-cases-testing-web-desktop-applications/


AW
Academy