SEIZEDATA

Unit Testing in Python

(파이썬을 이용한 단위 테스트)

씨즈데이터 개발실 박상훈

Contents

- 소프트웨어 테스트 및 단위 테스트
  - TDD: Test-Driven Development
- Python에서의 단위 테스트
  - unittest vs pytest
  - 단위 테스트의 작성
  - Given-When-Then pattern
  - Test data feeding: Fixture
  - Dependency handling: Mocking
  - 통합 테스트

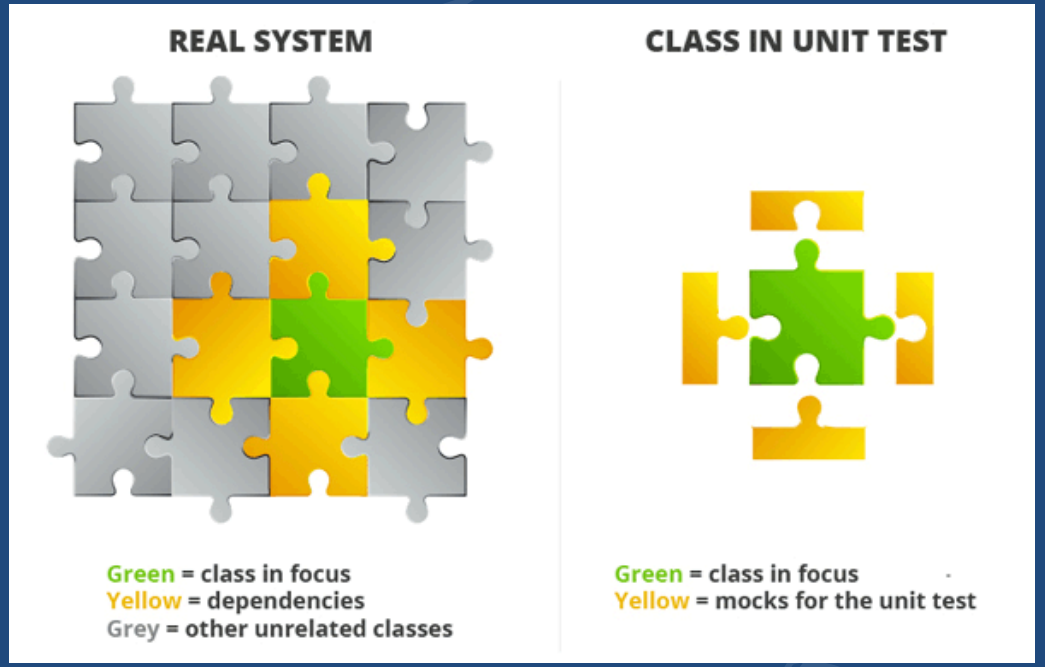
소프트웨어 테스트

- 테스트란?
  - 내가 작성한 코드가 제대로 동작하는지 확인하는 것

- 테스트의 종류
  - 단위 테스트 **(unit test)**
    : 개발자 뷰, 개별 기능(함수)에 대한 검증
  - 기능 테스트 **(function test)**
    : 사용자 뷰, 요구사항 단위
  - 통합 테스트 **(integration test)**
    : 모듈 간의 인터페이스 및 호환성을 검증
  - 인수 테스트 **(acceptance test)**
    : 실 사용자 환경에서, 사용자 입장으로 테스트
    ex) 클로즈 베타 테스트, 오픈 베타 테스트 등

단위 테스트

- 실제 시스템 vs 단위 테스트
  - 단위 테스트는 시스템 내의 (작은 단위의) 특정 기능에 대해 테스트하는 것
  - 하지만, 실제 시스템에서는 각 모듈간 의존성이 존재함
  - 그 외 테스트하기 어려운 경우 존재
    ex) DB, 네트워크 리소스 사용,
    외부 서비스, 과금 서비스 등
  - 의존성이 있는 기능에 대한 테스트 필요
    : "테스트 더블"
    ex) dummies, fakes, stubs, mocks



단위 테스트

- 단위 테스트(unit test)란?
  - 특정 모듈이 의도된 대로 정확히 작동하는지 검증하는 것
    - 가능한 작은 기능 단위로 모듈 작성 및 테스트
    - 가능한 모든 함수/메소드에 대한 테스트 케이스를 작성해야 함
  - 코드를 테스트 하는 코드를 작성
    - 테스트 코드를 테스트하는 코드는 작성할 필요 없음
  - 자동화하여 테스트를 실행
  - 전체 테스트 중 일부에 대해서만 별도 테스트를 수행할 수 있음

- 단위 테스트를 수행하면..? (단위 테스트에 대한 오해)
  - 개발 시간이 매우 길어진다?
  - 모든 버그를 없앨 수 있다?

단위 테스트

- 단위 테스트를 해야 하는 이유
  - 누군가는 반드시 작성한 코드를 테스트해야 한다
  - 실제 실행 환경을 구성하지 않아도 특정 기능에 대한 테스트를 수행할 수 있다
  - 소프트웨어의 버그를 줄일 수 있다 (고품질 소프트웨어)
  - 사용자의 입장에서 코드를 작성할 수 있다
    - 테스트 코드를 이용한 문서화 기능 (사용 설명서 / 의사 소통 수단)
  - 소프트웨어 설계를 개선할 수 있다
  - 코드를 보다 깔끔하고 컴팩트하게 유지할 수 있다
  - 기능 수정 시 심리적으로 편안하게(?) 작업할 수 있다
  - 개발 시간을 절약할 수 있다
  - 소프트웨어 유지/보수 비용을 절감할 수 있다

단위 테스트

- (자동화된) 단위 테스트가 없다면?
  - 자체 버그 검출 능력 저하
    : 모든 기능을 테스트 하기 어려움
  - 소스 코드의 품질 저하
    : 어디서 버그가 생길지 모르기 때문에
    잘못된 코드도 고치지 않으려고 함
  - 자체 테스트 비용의 증가
    : 작은 수정에도 모든 기능을 다시 테스트 해야 함

- 단위 테스트: ***"리팩토링 할 때 최소한의 버팀목"***

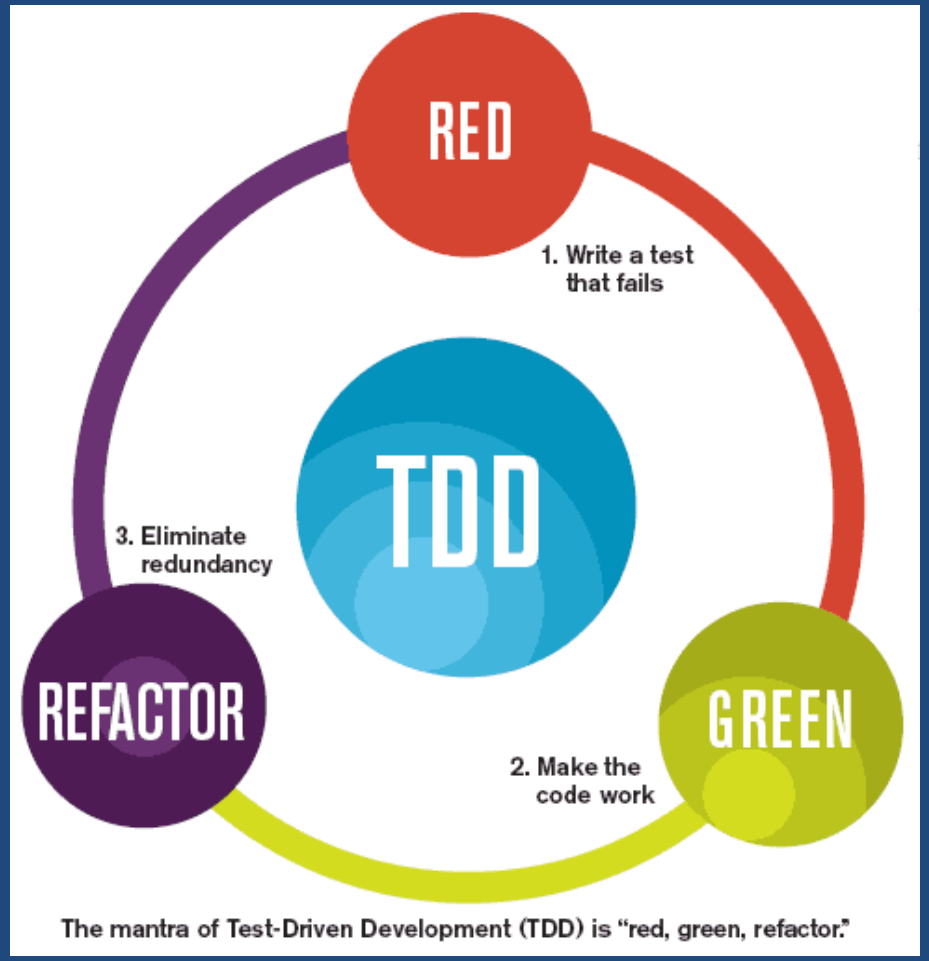
단위 테스트

- “테스트를 합시다”
  - 테스트는 개발의 (선택이 아닌) 필수적인 과정
    - 테스트 없이 고품질 코드를 오랫동안 유지하기 어려움
    - (물론 테스트가 있다고 코드의 질이 보장되지는 않음)
  - 사용자 관점에서 내가 작성한 코드를 사용하게 됨
  - 현재의 구현이 적절한지 다른 시각에서 판단 가능
  - 보다 효율적인 코드 구성이 가능

- 단위 테스트는 아래의 단계로 수행됨
  - 예상
  - 실행
  - 검증

Test-Driven Development (TDD)

- TDD: Test-Driven Development (테스트 주도 개발)
  - 코드가 가져야 할 스펙을 정의하고, 그에 맞는 테스트를 먼저 작성
  - 테스트 수행 후 실패하는 것을 확인
  - 실제 코드 작성 후, 테스트가 성공하는지 확인
- TDD의 원칙
  - 실패하는 단위 테스트를 작성할 때까지 실제 코드를 작성하지 않는다
  - 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성한다
  - 현재 실패하는 테스트 코드가 통과할 정도로만 실제 코드를 작성한다
- Demo
- 매우 촘촘한 테스트 케이스 작성 가능
- 그러나, 실제 적용 시 쉽지 않음...



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

단위 테스트에 대한 보다 쉬운 접근 (better than nothing)

- 코드를 작성할 때 테스트를 적극 활용
  - 작성 중인 개별 기능 확인을 위한 테스트 (프로덕션 환경 X)
- 버그를 찾으면 그에 해당하는 테스트 작성 (오답노트)
- 테스트를 작성하기 어려운 코드 (ex. 외부 리소스)
  - ~~그 코드에 대한 테스트 작성을 포기한다 (포기하면 편해...)~~
  - Mock / fake / stub / ... (이것도 쉽지는 않음..)
  - 코드 리팩토링 고려 (테스트 하기 쉬운 코드로 작성)
  - 테스트는 코드를 가장 적극적으로 활용하는 일종의 사용자

Python에서의 단위 테스트

- unittest
  - Python의 기본 테스트 프레임워크
  - Java의 JUnit이라는 테스팅 프레임워크로부터 영향
    - PEP 8 을 지키지 않음.. (e.g. unittest.TestCase, setUp(), assertEqual(), etc.)
    - 테스트를 작성할 때 반드시 클래스를 정의해야 함
  - 테스트 케이스 작성 및 실행 전반에 걸쳐 유용한 도구 제공
    : 테스트 케이스 검색, 테스트 사전/사후 작업 (fixture),
    테스트 케이스 실행, 테스트 결과 수집, 결과 보고 등등

- pytest
  - 테스트를 쉽게 할 수 있도록 도와주는 라이브러리
  - 성숙한 라이브러리
    - 2007년 1월 첫 커밋 / 최근까지 활발하게 커밋
  - unittest로 작성한 코드도 테스트 수행 가능
  - 독특하면서 강력한 fixture 문법 제공

unittest vs pytest

- unittest vs pytest

unittest:

```
from unittest import TestCase



class UpperTestCase(TestCase):
    def test_upper(self):
        self.assertEqual("foo".upper(), "FOO")
```

pytest:

```
def test_upper():
    assert "foo".upper() == "FOO"
```

- Demo

unittest

unittest: 테스트 에러 vs 에러

```python
def test_this_will_fail(self):
    self.assertEqual(myabs(5), 6)
```

```
FAIL: test_this_will_fail (test_code_unittest.TestMyAbs)
----------------------------------------------------------
Traceback (most recent call last):
  File
"/Users/adrysn/Develop/pycon17/test_code_unittest.py",
line 7, in test_this_will_fail
    self.assertEqual(myabs(5), 6)
AssertionError: 5 != 6

----------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

test exception

```python
def test_this_will_fail(self):
    assert myabs(5) == 6
```

```
FAIL: test_this_will_fail (test_code_unittest.TestMyAbs)
----------------------------------------------------------
Traceback (most recent call last):
  File
"/Users/adrysn/Develop/pycon17/test_code_unittest.py",
line 7, in test_this_will_fail
    assert myabs(5) == 6
AssertionError

----------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

python exception

그래서 unittest 모듈 이용할 경우 assert*() 메소드 사용을 권장

unittest

다양한 assertion 메소드

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Method	Checks that
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	a and b have the same elements in the same number, regardless of their order

Method	Used to compare
assertMultiLineEqual(a, b)	strings
assertSequenceEqual(a, b)	sequences
assertListEqual(a, b)	lists
assertTupleEqual(a, b)	tuples
assertSetEqual(a, b)	sets or frozensets
assertDictEqual(a, b)	dicts

Method	Checks that
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises exc
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises exc and the message matches regex r
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises warn
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises warn and the message matches regex r
assertLogs(logger, level)	The with block logs on logger with minimum level

단위 테스트의 작성

- 단위 테스트 작성 방법
  - 구현 기능에 대한 테스트 코드를 작성
    (i.e. 구현한 함수를 호출하는 테스트 코드를 작성)
  - 기대하는 (알고 있는) 결과값을 이용하여 assert 구문 작성 [예상]
  - 테스트 수행 후 성공 여부를 확인 [실행 / 검증]
  - 모든 테스트 모듈 및 메소드 이름은 "test"로 시작하도록 명명
    - pytest (혹은 unittest)가 자동으로 테스트 메소드 탐색 가능
  - 메소드명을 보고 어떤 테스트 케이스인지 이해할 수 있도록 작성
  - "edge cases"에 대한 테스트 케이스를 잘 작성하는 것을 권장

단위 테스트의 작성

- 단위 테스트 작성 원칙
  - 한 번에 하나만 테스트 해야 함
  - 실패가 명확해야 함
  - 빠르게 테스트 되어야 함
  - 자동화 되어야 함
  - 중복되지 않아야 함
  - 각자의 테스트는 서로 독립적이어야 함
    - 테스트의 결과는 각 테스트의 수행 순서와는 무관해야 함
    - 개별 테스트 혹은 테스트의 일부를 별도로 수행할 수 있어야 함

Given-When-Then Pattern

- Given-When-Then pattern (arrange-act-assert or preconditions-trigger-postconditions)
  - Given: 테스트 전의 상태 (== arrange == preconditions)
  - When: 테스트 행위 (== act == trigger)
  - Then: 테스트 검증 (== assert == postconditions)

  - Example / Demo

```
Feature: User trades stocks(사용자 주식거래)
  Scenario: User requests a sell before close of trading
    Given I have 100 shares of MSFT stock
       And I have 150 shares of APPL stock
       And the time is before close of trading
(사용자가 거래를 마감하기 전에 판매를 요청
MSFT 주식 100주를 가지고 있는 걸 감안
그리고 APPL 주식 150주를 보유
그리고 그 시간은 거래가 끝나기 전)


When I ask to sell 20 shares of MSFT stock
(MSFT 주식 20주를 판매)


     Then I should have 80 shares of MSFT stock
      And I should have 150 shares of APPL stock
      And a sell order for 20 shares of MSFT stock should have been
executed
(그럼 MSFT 주식 80주를 가져야겠다.
그리고 APPL 주식 150주를 가져야 한다.
그리고 MSFT 주식 20주에 대한 매도 주문이 실행되었어야 했다.)
```

Fixture

- 만약 테스트를 위해 리소스(데이터)가 필요하다면..?
  -> Fixture

- unittest
  - setUp() / tearDown() 메소드
    : 모든 테스트의 시작/종료 시 수행되는 코드
  - setUpClass() / tearDownClass() 메소드
    : 해당 클래스의 테스트 시작/종료 시 한 번만 수행되는 코드
  - Demo

Fixture

- Fixtures in “pytest”
  - 클래스 없이 함수만으로 fixture 및 테스트 케이스 작성 가능
  - 테스트에 필요한 공통 자원
  - @pytest.fixture 데코레이터를 이용하여 fixture 선언
    - 데이터를 함수 형태로 작성
  - 선언한 fixture를 테스트 메소드의 인자로 전달
  - 테스트 케이스에서 필요로 하는 fixture를 선택적으로 쉽게 사용
    - Dependency injection 형태
  - 테스트의 독립성과 자원 재사용 사이의 조화를 찾는 한 방편

- pytest fixture의 장점
  - Fixture 관련 다양한 추가 기능들 제공
    : finalization / parametrization / fixture가 다른 fixture 사용 가능 / fixture over-riding 가능
  - 다양한 플러그인 제공

Fixture

- Fixtures in “pytest”

```python
# content of ./test_smtpsimple.py
import pytest


@pytest.fixture
def smtp_connection():
    import smtplib

    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)


def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert 0  # for demo purposes
```

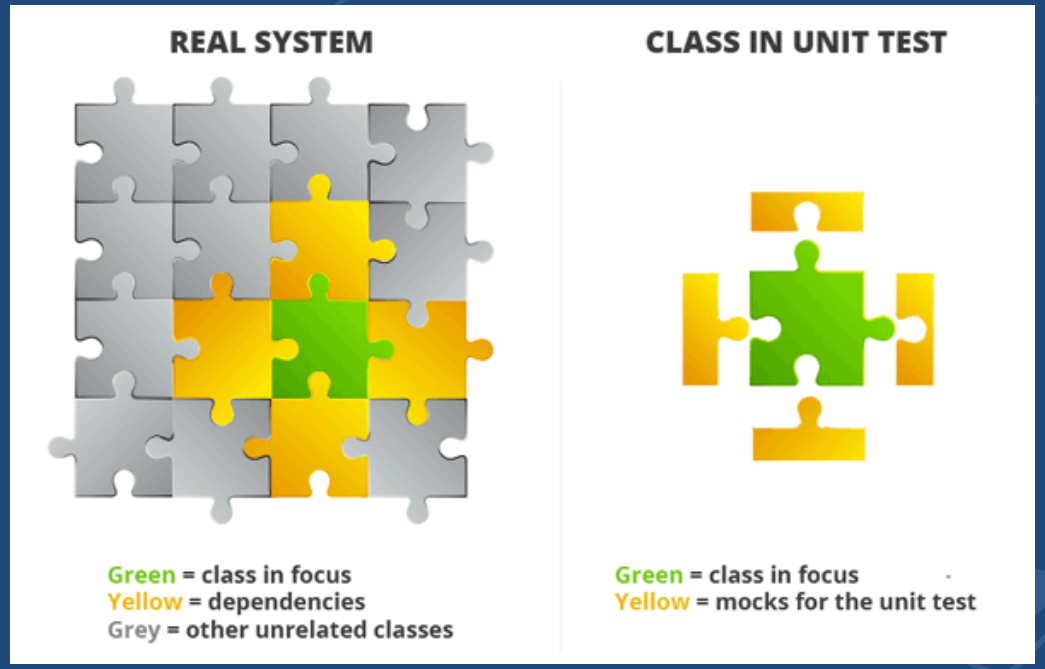
```
$ pytest test_smtpsimple.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_smtpsimple.py F                                                  [100%]

================================ FAILURES ================================
_____________________________ test_ehlo _____________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_smtpsimple.py:14: AssertionError
========================== 1 failed in 0.12s ==========================
```

- Demo

Mock

- 의존성이 있는 기능에 대한 테스트가 필요하다면..?
  -> Mock

- Mock 대상이 되는 것들
  - 시간이 오래 걸리는 것
  - 값이 변하는 것
  - 상태가 유지되는 것
    (다른 테스트에 영향을 주는 것)
  - 시스템 콜
  - 네트워크로 연결된 것
  - 준비하기 복잡한 것
  - 과금이 되는 것



Mock

- 의존성이 있는 것들을 실제로 실행시키지 말고,
  호출 여부, 인터페이스만 확인하자
  - Mock 객체로 실제 객체를 대체
  - 해당 객체를 이용하여
    함수 호출 여부 및 호출 인자만 확인

- Mock 패키지 이용
  - 파이썬 기본 라이브러리에 포함 (Python>=3.3)
    ```
    >>> from unittest import mock
    ```
  - 직접 설치
    ```
    $ pip install mock
    ```

- Monkey Patch: 런타임에 클래스, 함수 등을 테스트 코드로 대체하는 것

- Demo

통합 테스트

- 통합 테스트 **(Integration Test)**
  - 다른 모듈, 서비스 등을 붙여서 그 관계에서의 문제점을 확인하는 과정

- 통합 테스트의 어려운 점
  - 개발 환경과 프로덕션 환경의 차이?
  - 배포 환경에서 동시 테스트가 제한적인 경우?
  - “나만 쓰는” 프로덕션 환경과 동일한 테스트 환경이 있었으면...

  -> Docker 이용 (Linux Container / light VM)

  - **UI** 동작 테스트.. 직접 손으로?

  -> Selenium 이용 (웹 브라우저 자동화 툴)

덧붙이는 사항들

- 테스트의 범위는?
  - 최소: 테스트 케이스가 메인 로직을 검증
  - 최대: 개발 시간의 2배 이상을 쓰지 않도록

  -> 이 사이에서 가장 가성비 높은 지점을 찾자!

- 테스트의 시행 시점은?
  - 개발할 때
  - commit할 때
  - push할 때
  - CI 서버에서
  - etc.

Just Do It !

테스트는 힘들지만
가치 있는 일입니다

지금 시작하세요!

References

- 참고 자료
  - Python 테스트 시작하기 - 이호성 [PyCon KOREA 2015]
    - 영상: https://www.youtube.com/watch?v=hAUjltE42cY
    - 슬라이드: https://www.slideshare.net/hosunglee948/python-52222334
  - 파이썬에서 편하게 테스트 케이스 작성하기: pytest, Travis CI, 그리고 도커 - 박종현 [PyCon KOREA 2017]
    - 영상: https://www.youtube.com/watch?v=rxCjxX4tT1E
    - 슬라이드: https://archive.pycon.kr/2017/program/132
- 유용한 사이트 및 읽을 거리
  - 단위 테스트 / TDD / BDD: https://feel5ny.github.io/2017/12/08/TDD_01/
  - unittest vs pytest: https://www.bangseongbeom.com/unittest-vs-pytest.html
  - 테스트 더블: https://brunch.co.kr/@tilltue/55
  - 코드 커버리지 80% 넘긴 썰: https://brunch.co.kr/@leehosung/43