

Unit Test의 이해

CONTENTS

목차

Chapter1

Unit Test 개요

Chapter2

pytest 해보기

Chapter3

pytest 기능 활용

Chapter4

Test Fixture

Chapter5

pytest etc

Chapter6

Unit Test 의 의미

Chapter 1

Unit Test 개요

Unit Test 란?

- 작은 단위 (주로 함수 or 클래스 단위)로 Testing을 하는 것
입력값을 넣어 보았을 때,
결과값이 기대값과 동일하게 나오는지 확인한다.

Python 진영에서 가장 많이 사용되는 Unit Test Framework

- 파이썬에는 "unittest" 라는 Unit Test Framework가 기본적으로 제공되지만 파이썬 진영에서는 pytest 라이브러리를 더 많이 사용함
- 별도 설치 필요
 - `pip install pytest`
- pytest 3.8 이상 버전에서 동작

pytest 사용 가이드 (Github & Document)

- <https://docs.pytest.org/en/stable/>
- <https://github.com/pytest-dev/pytest>

unittest VS pytest

unittest

- 파이썬 표준 Library에 포함되어 있음
 - 별도 설치 필요 없음
- xUnit 스타일의 코드 (gTest, JUnit 등)
 - xUnit 스타일의 UnitTest 경험자들은, 손쉽게 손쉽게 사용 가능

pytest



- 2010년 10월 공식 버전 출시
- 더 파이썬스러운 테스트 코드 작성 가능
 - 더 파이썬스러운 코드로 테스트 진행
- 별도 설치 필요
 - UnitTest : pip install pytest
 - Mocking : pip install pytest-mock

Chapter 2

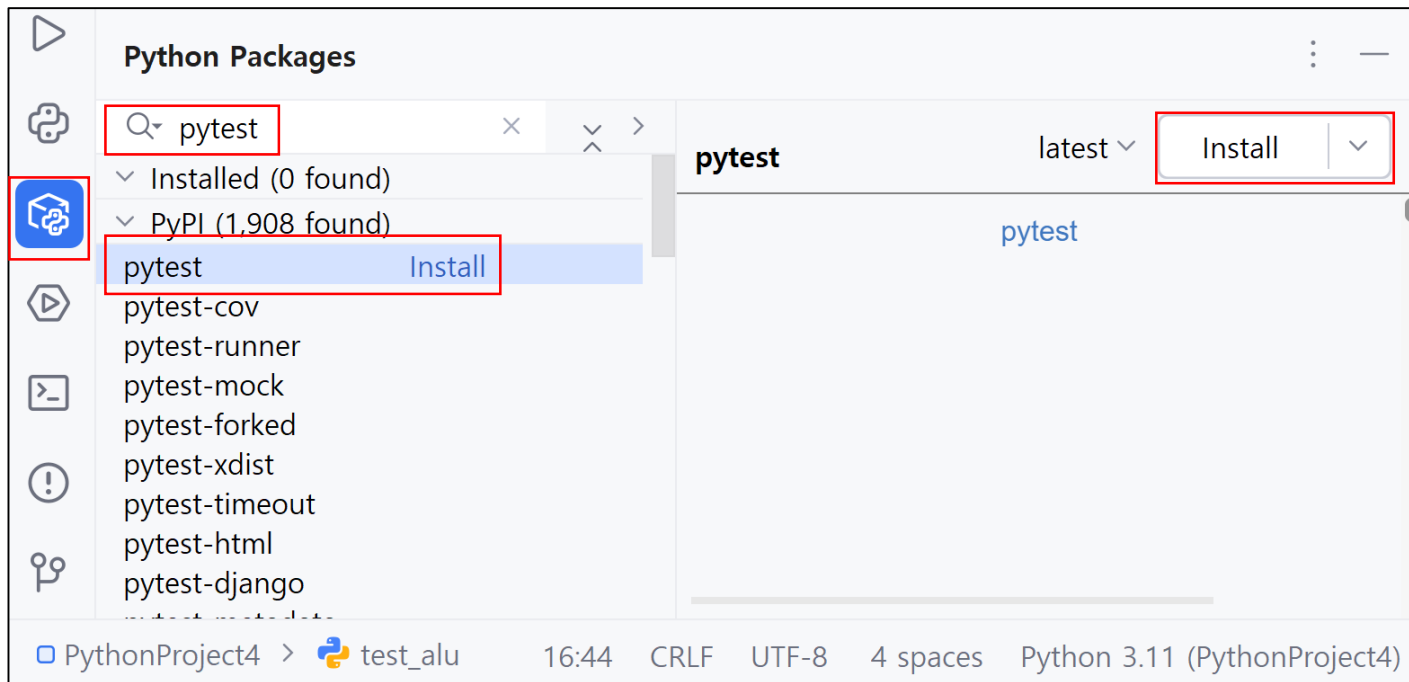
pytest 해보기

pytest 를 통한 유닛테스트 코드 작성

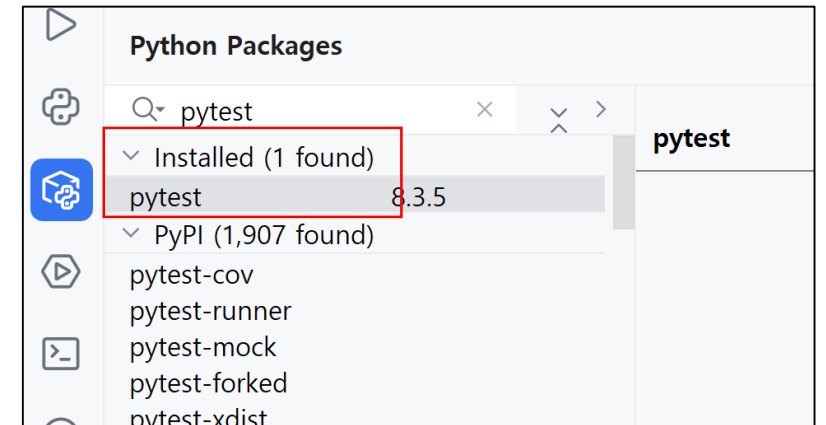
pytest 설치

pytest 패키지를 설치한다

- pytest : 유닛테스트 패키지



pytest 클릭 후
install 버튼을 눌러 설치

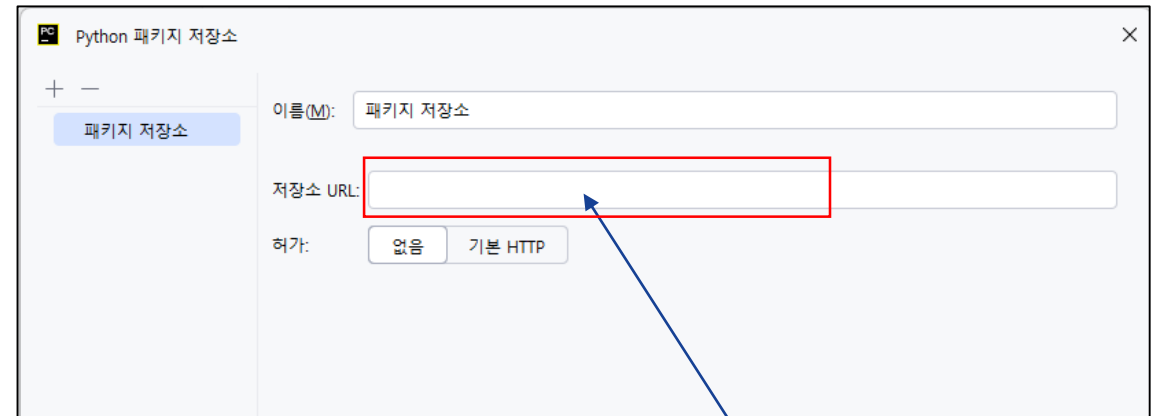
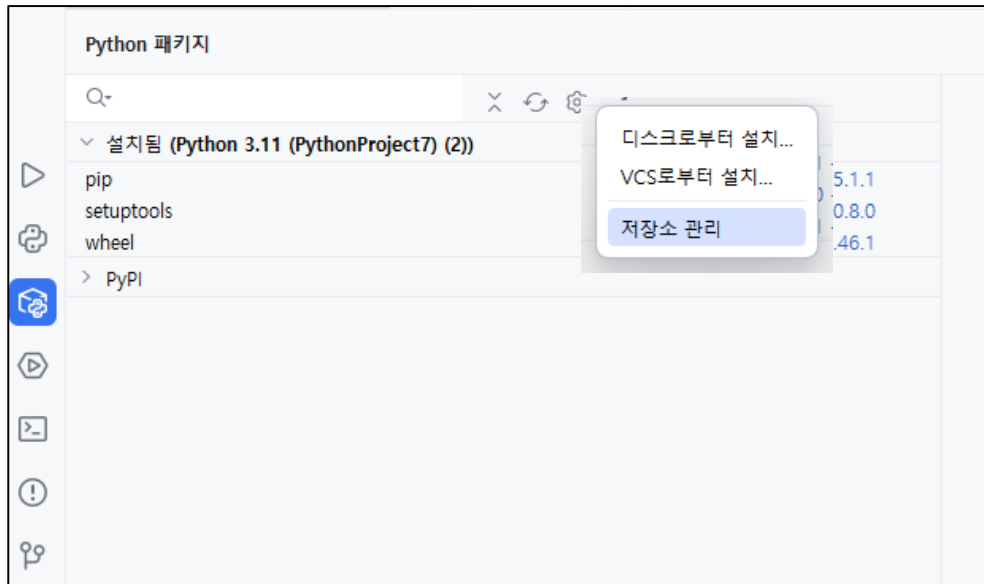


설치가 되었음을
이곳에서 확인할 수 있다.

[Trouble Shooting] python 패키지 저장소 설정

pytest 검색이 안되는 경우, 저장소 설정을 해본다.

pycharm 하단부에서 패키지 설치메뉴 > 저장소 연동



<https://pypi.org/simple>
기입하기

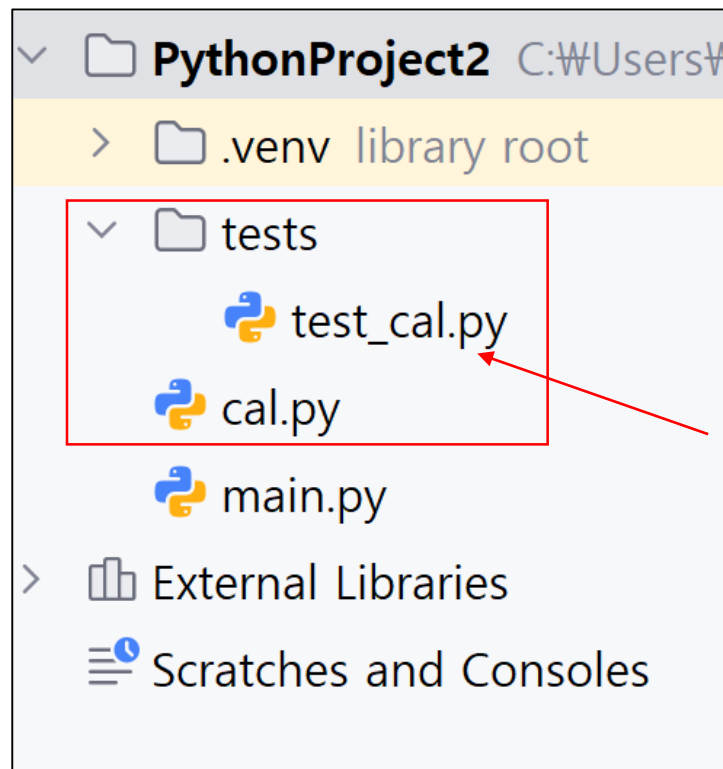
pytest 세팅하기

Directory를 추가한다.

- tests 폴더 : 테스트 파일을 넣는 곳

2개의 py 파일을 생성한다.

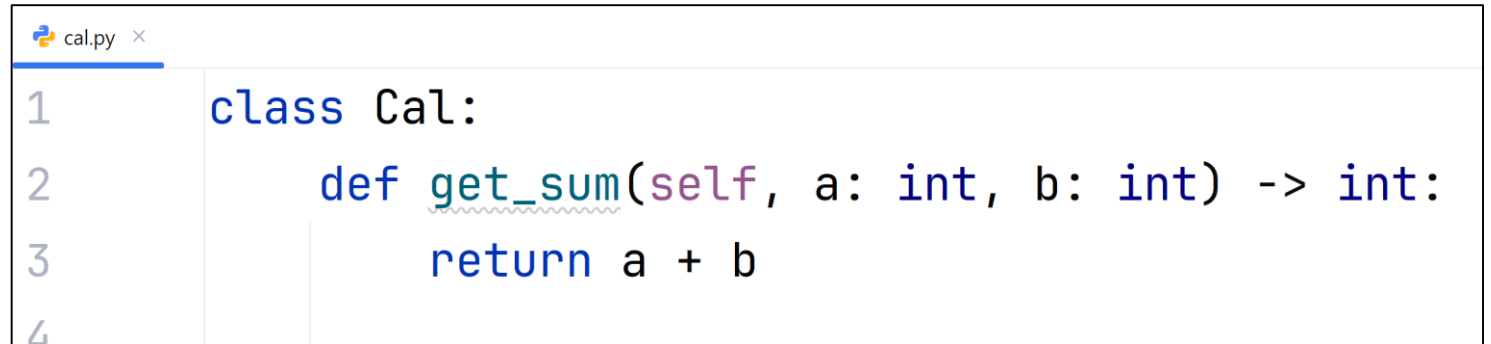
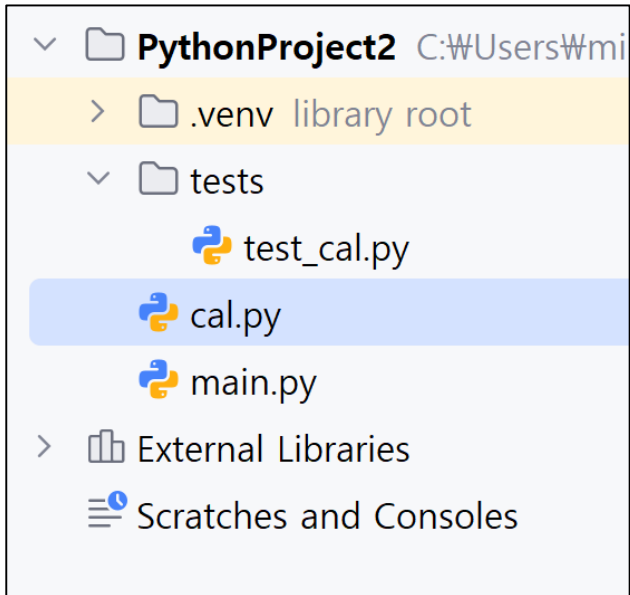
- cal.py 파일을 생성
- test_cal.py 파일을 tests 폴더에 생성



테스트 파일명은
"test_" 로 시작해야
테스트 파일로 인식
함

테스트 대상이 될 소스코드 입력하기

cal.py 파일에 기본 코드를 입력한다.

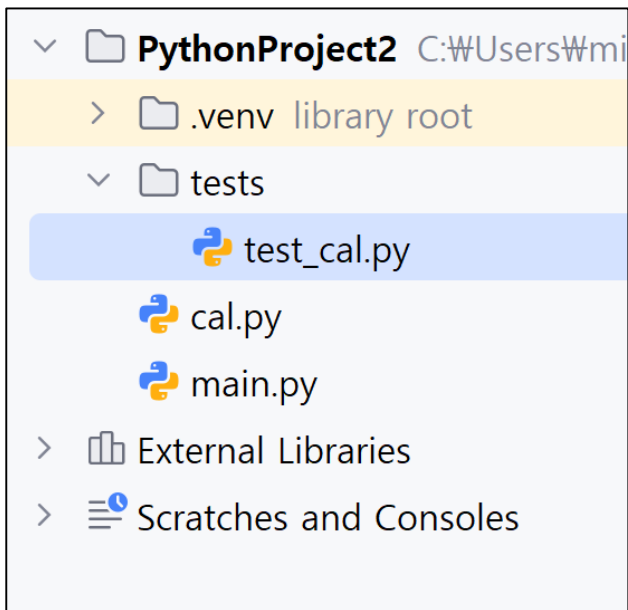


A screenshot of a code editor window titled 'cal.py'. It displays the following Python code:

```
1 class Cal:
2     def get_sum(self, a: int, b: int) -> int:
3         return a + b
4
```

테스트 코드 삽입

테스트 코드 삽입하기



pytest 임을 명시하기 위해
항상 import pytest를 써주자

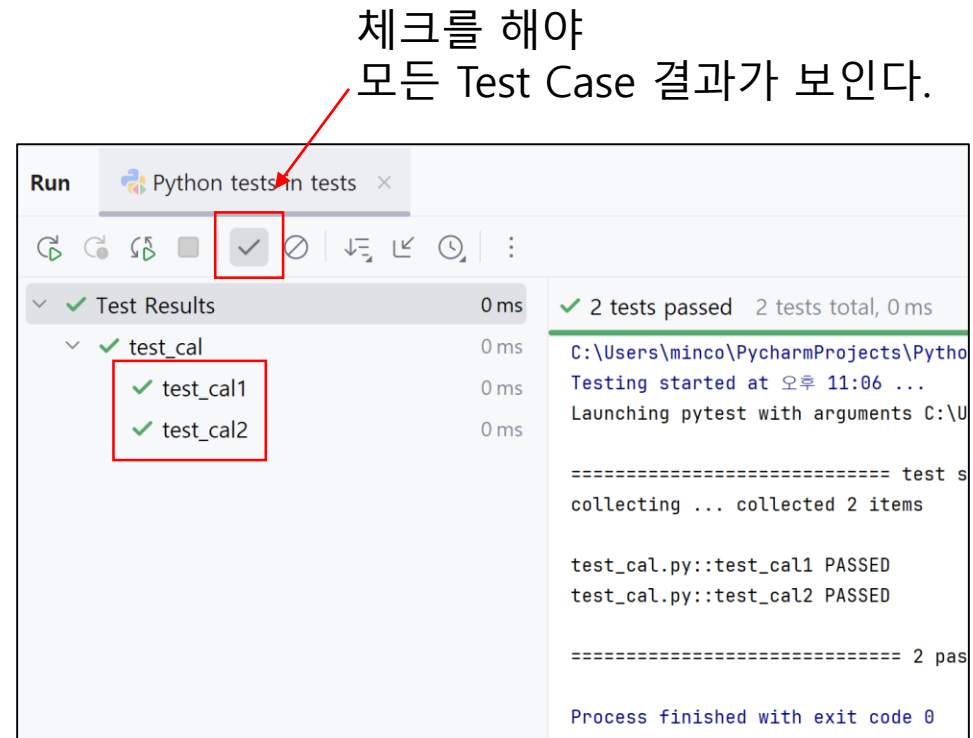
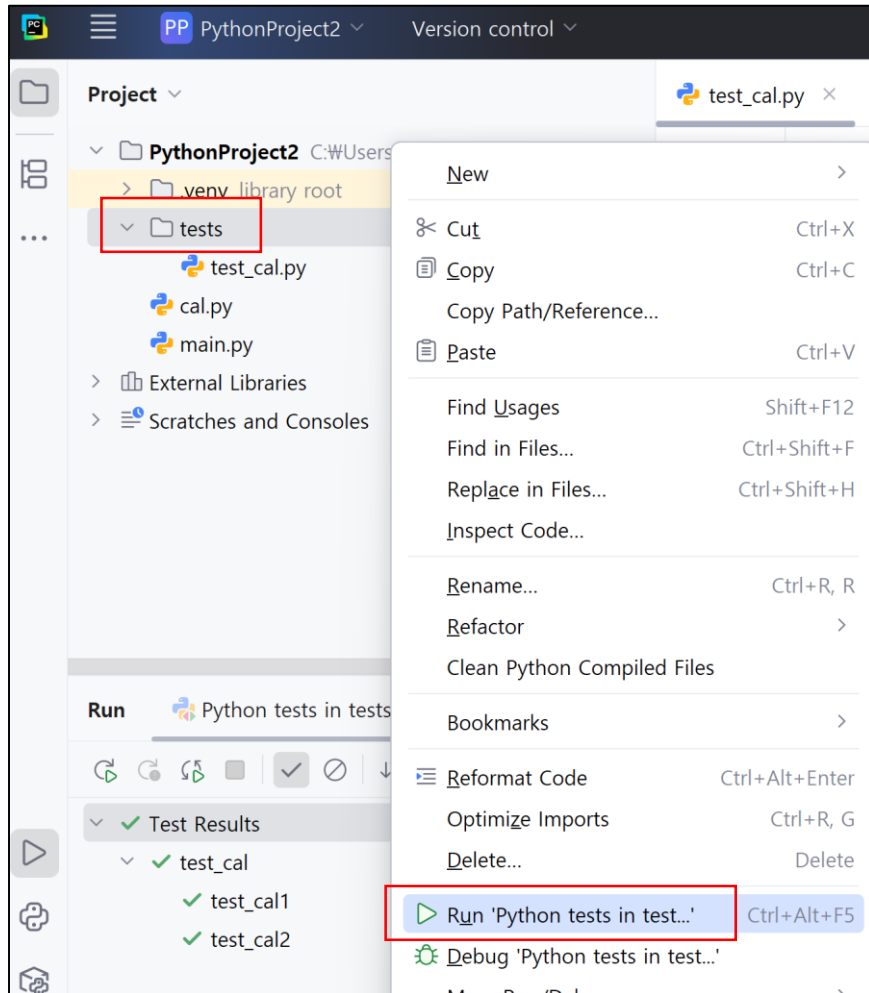
cal.py 파일

Cal 클래스

```
import pytest
from cal import Cal
def test_cal1():
    cal = Cal()
    result = cal.get_sum(1, 2)
    assert result == 3
def test_cal2():
    cal = Cal()
    result = cal.get_sum(10, 20)
    assert result == 30
```

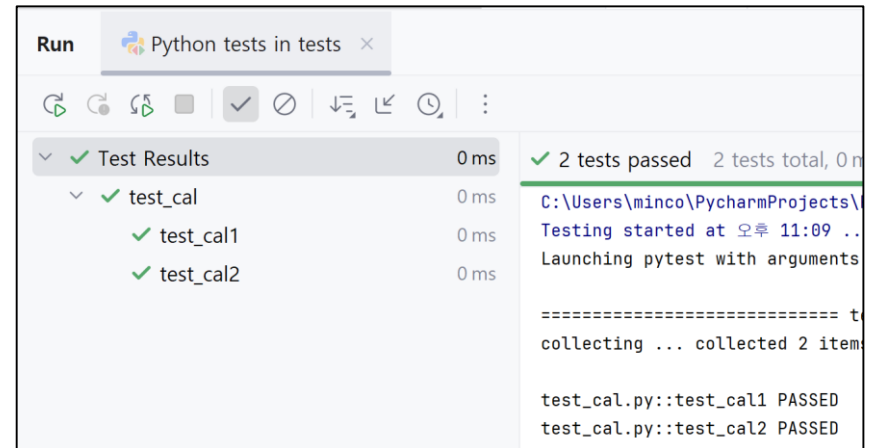
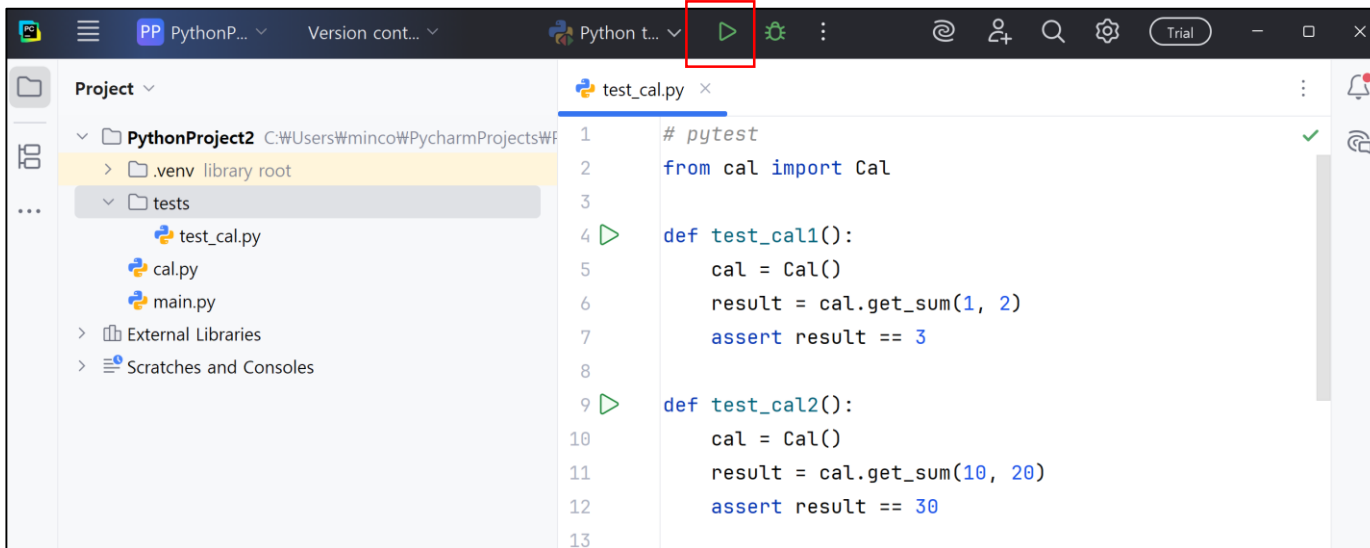
테스트 RUN - 1회성 세팅

tests 폴더로 테스트를 하면, 모든 테스트를 수행할 수 있다.



테스트 RUN

이후에는 RUN 버튼만 누르면
모든 테스트가 수행된다.



유닛테스트 구성

먼저 유닛테스트를 실행해보고 구성을 살펴본다.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

        with self.assertRaises(TypeError):
            s.split(2) # ----> separator는 string 이어야 정상 동작
```

test_string_methods.py

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/unittest/basic_example.py

pytest 로 변경하기

앞에서 진행했던 코드는 python unittest 로 작성된 코드이다.
이를 pytest 설치 후 변경을 해보자.

pytest 코드로 변경

다음과 같이 pytest 코드를 작성한다

```
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()

def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']

    with pytest.raises(TypeError):
        s.split(2) # separator는 string이어야 정상 동작
```

python unittest VS pytest

두 코드를 비교해보며 pytest 작성법을 확인한다.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

        with self.assertRaises(TypeError):
            s.split(2) # ----> separator는 string 이어야 정상 동작
```

python unittest 코드

```
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()

def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']

    with pytest.raises(TypeError):
        s.split(2) # separator는 string 이어야 정상 동작
```

pytest 코드

테스트 코드 구현하기

1. 테스트 함수

이름이 test_ 로 시작해야 pytest 가 자동으로 인식

2. Assertion

unittest 의 self.assertEqual() 대신 Python 내장 assert 사용
실패시 pytest 기능으로 차이점 출력

```
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()

def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']

    with pytest.raises(TypeError):
        s.split(2) # separator는 string이어야 정상 동작
```

테스트 수행하기

실패하는 테스트 추가 후 출력 메시지를 확인한다

```
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()

def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']

    with pytest.raises(TypeError):
        s.split(2) # separator는 string이어야 정상 동작

def test_fail():
    assert 1 == 2
```

```
===== test session starts =====
collecting ... collected 4 items

test_something.py::test_upper PASSED [ 25%]
test_something.py::test_isupper PASSED [ 50%]
test_something.py::test_split PASSED [ 75%]
test_something.py::test_fail FAILED [100%]
test_something.py:16 (test_fail)
1 != 2

필요:2
실제 :1
<클릭하여 차이점 확인>

def test_fail():
> assert 1 == 2
E assert 1 == 2

test_something.py:18: AssertionError

===== 1 failed, 3 passed in 0.10s =====
```

E : 오류 메시지, assert 실패 내용
> : 실패한 코드 라인

[도전] 새 프로젝트 부터, 처음부터 만들기

Gop 함수 생성하기

Calculator 클래스에 두 수의 곱을 리턴해주는 함수를 구현한다.

Unit Test하기

Gop 함수가 정상 동작하는지 테스트
테스트 케이스를 2개 만든다.

최대한 처음부터 힌트없이 만들어본다.

Given-When-Then 단계를 이용하면 각 단계별 코드들의 의도를 파악하기 쉽다

Arrange (Given): 테스트의 전제조건

테스트를 위한 준비단계로 필요한 객체 생성, 초기 상태 설정 등의 코드가 들어간다.

Act (When) : 테스트의 핵심행동

실제 테스트 대상(SUT)의 동작 수행을 한다.

Assert (Then) : 검증 목적

결과를 검증하는 단계로 pytest 에서는 assert 를 사용한다

[도전] AAA 패턴으로 작성하기

transfer_to 메서드 테스트코드 작성하기

다음은 Account 클래스의 계좌 이체함수 기능의 코드이다.

1. 계좌 이체 시 송신측 예금액이 차감되고 수신측 계좌 예금액은 증가한다.
2. 만약 예금이 이체 금액보다 부족한 경우 ValueError가 발생한다.

Mission

두 기능을 테스트할 수 있는 테스트케이스를 AAA 패턴으로 작성해본다.

```
class Account:
    def __init__(self, balance):
        self.balance = balance

    def transfer_to(self, target, amount):
        if self.balance < amount:
            raise ValueError("예금 부족")
        self.balance -= amount
        target.balance += amount

# 간단한 계좌 이체 함수를 pytest와 함께 AAA 패턴으로 작성해본다
```

bank.py

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/refactoring/aaa_patern_write_test.py

Chapter 3

pytest 기능 활용

내장 assert 문

내장 assert 문을 사용해서 검증할 수 있지만 실패 메시지가 부실하다.

```
▶ class AssertTestCase(unittest.TestCase):  
▶ def test_assert_statement(self):  
    expected = 12  
    actual = 2 * 5  
    assert expected == actual
```

```
Failure  
Traceback (most recent call last):  
  File "C:\reviewer\PythonProject2\test.py", line 8, in test_assert_statement  
    assert expected == actual  
    ^^^^^^^^^^^^^^^^^^^^^^^^^  
AssertionError  
  
Ran 1 test in 0.007s  
  
FAILED (failures=1)
```

왜 실패했는지, 값을 확인 할 수 없다.

pytest 에서 assert 문

pytest 는 assert re-writing 으로 단순히 AssertionError가 아니라 상세한 오류내역을 출력해준다.

```
def test_assert():  
    expected = 12  
    actual = 2 * 5  
    assert expected == actual
```

```
12 != 10  
  
Expected :10  
Actual   :12  
<Click to see difference>  
  
def test_assert():  
    expected = 12  
    actual = 2 * 5  
>    assert expected == actual  
E    assert 12 == 10  
  
main.py:21: AssertionError
```

pytest 의 assert 사용방법

다음 assert 표현식을 보고 해석해본다.

```
assert expr
assert not expr
assert a == b
assert a != b
assert a is None
assert a is not None
assert a > b
assert a <= b
```

예외 테스트

pytest.raises 를 통해 with 안의 코드 블록이 **예외를 발생시키는지** 검증한다.

```
import pytest

def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        a = 1 / 0
```

ZeroDivisionError 가 발생하는지 체크한다

pytest.fail()

pytest.fail() : 명시적 테스트

assert 를 사용하기 애매한 복잡한 조건일 때, **명시적인 테스트 실패 기능** 조건문으로 판단 후 실패코드가 실행되게 한다.

pytest.fail 활용

복잡한 조건을 검사하거나, 여러 단계를 거친 후 실패 처리를 하고 싶을 때 사용한다.

```
def test_conditions():  
    result = some_function()  
    if result.status == "ok" and result.value is None:  
        pytest.fail("status 가 ok 인경우 value는 None 이 아니어야 한다")
```

Chapter 4

Test Fixture

pytest 의 fixture 기능으로 테스트 코드 관리

fixture로 공통 설정 제공

@pytest.fixture 를 사용해서 공통된 설정을 제공할 수 있다.

* fixture : 반복적인 테스트환경을 구축하도록 도와준다. (고정장치 실험도구에서 유래)

```
import pytest

@pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    return "chicken"

def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")

def test_kfc(ingredient):
    assert ingredient == "chicken"
    print("## KFC 치킨 요리 ##")
```

Setup 동작

매 Test Case마다 수행되는
동작을 **Setup** 이라고 한다.

```
db.py::test_bbq
## 닭 준비 ##
## BBQ 치킨 요리 ##
PASSED
db.py::test_kfc
## 닭 준비 ##
## KFC 치킨 요리 ##
PASSED
```


fixture 실행 순서

1. test_method(fixture) 호출할때, 먼저 fixture 함수가 실행된다.
2. fixture 함수의 반환값이 test_method(fixture)의 인자로 주입된다.
3. 그 후 test_method 의 내용이 실행된다.

```
@pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    return "chicken"

1 def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")
```

test_bbq 실행 순서가 되면
ingredient가 필요하다는 것을 인지한다.

```
2 @pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    return "chicken"

def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")
```

ingredient 함수에서 값을 반환한다.

```
@pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    return "chicken"

3 def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")
```

ingredient 객체가 준비가 되었고
테스트 함수를 수행한다.

fixture에 teardown 도입

Test Case 종료시마다 동작되는 코드 도입하기

- Setup : Test Case **시작**시 마다 동작되는 코드, Fixture에 추가하면 된다.
- **TearDown** : Test Case **종료**시 마다 동작되는 코드, Fixture에 추가하면 된다.

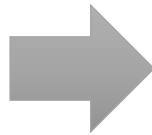
Setup

```
import pytest

@pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    return "chicken"

def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")

def test_kfc(ingredient):
    assert ingredient == "chicken"
    print("## KFC 치킨 요리 ##")
```



Setup

Teardown

```
import pytest

@pytest.fixture
def ingredient():
    print()
    print("## 닭 준비 ##")
    yield "chicken"
    print()
    print("요리 끝, 청소")

def test_bbq(ingredient):
    assert ingredient == "chicken"
    print("## BBQ 치킨 요리 ##")

def test_kfc(ingredient):
    assert ingredient == "chicken"
    print("## KFC 치킨 요리 ##")
```

```
db.py::test_bbq
## 닭 준비 ##
## BBQ 치킨 요리 ##
PASSED
요리 끝, 청소

db.py::test_kfc
## 닭 준비 ##
## KFC 치킨 요리 ##
PASSED
요리 끝, 청소
```

fixture의 활용 예시

매 Test Case마다 반복적인 초기화 작업 or 초기화 + 정리 작업이 필요할 때

Setup : in-memory DB 생성코드

Tearown : in-memory DB 제거코드

```
import sqlite3

def connect_db():
    # 메모리 DB와 테이블 생성
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE chicken (name TEXT)")
    conn.commit()
    return conn

def disconnect_db(conn):
    # DB Close와 함께 메모리 DB 소멸됨
    conn.close()
```

```
import pytest
from db import connect_db, disconnect_db

@pytest.fixture
def db_conn():
    conn = connect_db()
    yield conn
    disconnect_db(conn)

def test_bbq(db_conn):
    # db_conn을 이용한 DB Query 테스트
    print("## DB 테스트 1 ##")

def test_kfc(db_conn):
    # db_conn을 이용한 DB Query 테스트
    print("## DB 테스트 2 ##")
```

Setup

Teardown

fixture - 임시디렉토리 제공하기

각 테스트별로 디렉토리가 필요한 경우, 임시디렉토리를 제공하여 테스트에 사용 후 제거할 수 있다

```
@pytest.fixture
def temp_workspace(tmp_path):
    # fixture 의 인자로 tmp_path 를 써준다

    workspace = tmp_path / "tmp_workspace"
    workspace.mkdir()
    print(f"\n디렉토리 생성: {workspace} #####")

    yield workspace

    print(f"\n정리 : {workspace} 자동 정리됨#####")
```

```
def test_create_file(temp_workspace):
    file_path = temp_workspace / "example.txt" # 파일 생성

    file_path.write_text("HIHI")

    assert file_path.exists()
    assert file_path.read_text() == "HIHI"
```

```
test_something.py::test_create_file
디렉토리 생성: C:\Users\jeong\AppData\Local\Temp\pytest-of-jeong\pytest-3\test_create_file0\tmp_workspace #####
PASSED [100%]
정리 : C:\Users\jeong\AppData\Local\Temp\pytest-of-jeong\pytest-3\test_create_file0\tmp_workspace 자동 정리됨#####
```

[참고] tmp_path 기본 사용법

파일 만들기

```
file = tmp_path / "file_name.txt"
```

파일 쓰기

```
file.write_text("hello world")
```

파일 읽기

```
my_text = file.read_text()
```

파일 존재 확인하기

```
file.exists()
```

```
def test_tmp_path(tmp_path):  
    file = tmp_path / "this_is_file.txt"  
    file.write_text("hello")  
    my_text = file.read_text()  
  
    print(my_text)
```

테스트 함수들 모으기

테스트 함수들을 하나의 클래스로 모아 그룹핑을 할 수 있다

TestXXX클래스를 만들어서 관련된 테스트 함수들을 하나의 단위로 묶을 수 있다

```
class TestString:
    def test_upper(self):
        assert 'foo'.upper() == 'FOO'

    def test_isupper(self):
        assert 'FOO'.isupper()
        assert not 'Foo'.isupper()

    def test_split(self):
        s = 'hello world'
        assert s.split() == ['hello', 'world']

        with pytest.raises(TypeError):
            s.split(2)
```

[도전] 홀짝 Unit Test

List 에 숫자 값을 넣으면,

짝수면 "O" / 홀수면 "X"를 구분해주는 List를 리턴하는 모듈

입력예시 : [1, 2, 3, 0]

출력예시 : ['X', 'O', 'X', 'O']

모두 짝수이거나 홀수면 null 리턴

시작 코드만 만들고,

테스트 코드를 만들자.

그리고 시작 코드를 완성하여 테스트한다.

```
1 class OddEven:
2     def get_result(self, nums):
3         result = []
4         return result
5
```

시작코드

Chapter 5

pytest etc

parameterized test

파라미터화 테스트

동일한 테스트 로직으로 다양한 테스트 데이터 조합을 반복 실행하게 할 수 있다

```
@pytest.mark.parametrize("입력값들", [test_case들])  
def test_함수(입력값들):  
    # test_case들을 read 할 수 있다  
    ...
```

첫 번째 인자는 파라미터 이름(문자열 or 리스트)
두 번째 인자는 리스트 of 테스트 케이스



test_함수 case 1

test_함수 case 2

test_함수 case 3

test_함수 case 4

test_함수 case 5

parametrize test 예시

```
@pytest.mark.parametrize("a,b,c,d", [(1,2,3,4),(5,6,7,8)])
def test_sample(a,b,c,d):
    print("\n#####")
    print(a,b,c,d)
    print("#####")
```

```
PASSED [ 50%]
#####
1 2 3 4
#####
PASSED [100%]
#####
5 6 7 8
#####
```

```
@pytest.mark.parametrize("dict_case", [
    {"id" : 1, "name" : "Minco"},
    {"id" : 2, "name" : "CoCo"}
])
def test_sample(dict_case):
    print()
    print(f'{dict_case["id"]} : {dict_case["name"]}')

```

```
PASSED [ 50%]
1 : Minco
PASSED [100%]
2 : CoCo
```

[도전] 덧셈 함수 테스트

파라미터화 테스트를 이용해서 3개의 TestCase 를 작성한다.

- $1 + 2 = 3$
- $3 + 4 = 7$
- $-1 + -1 = -2$

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

덧셈함수를 테스트하는 3개의 테스트케이스를 하나의 코드로 만든다

pytest 마커 기능

@pytest.mark.skip : 무조건 생략

```
@pytest.mark.skip("아직 미구현")
def test_feature():
    ...

def test_something():
    assert 1 == 1
```

✓ Test Results	0 ms	✓ 1 test passed, 1 ignored 2 tests total, 0 ms
✓ test_something	0 ms	C:\reviewer\PythonProject12\.venv\Scripts\python.exe "C
○ test_feature	0 ms	Testing started at 오후 4:00 ...
✓ test_something	0 ms	Launching pytest with arguments C:\reviewer\PythonProje
		===== test session starts =====
		collecting ... collected 2 items
		test_something.py::test_feature SKIPPED (아직 미구현)
		Skipped: 아직 미구현
		test_something.py::test_something PASSED

capsys fixture

pytest모듈에 내장된 capsys 는 stdout, stderr 를 캡처하는 데 사용한다.
CLI 로 출력되는 내용을 검증할 수 있다

```
def test_capsys(capsys):  
    print("hello") # 콘솔로 출력 되는 내용 : "hello\n"  
    captured = capsys.readouterr()  
    assert captured.out == "hello\n"
```

 capsys.readouterr() 의 out / err 을 이용한다

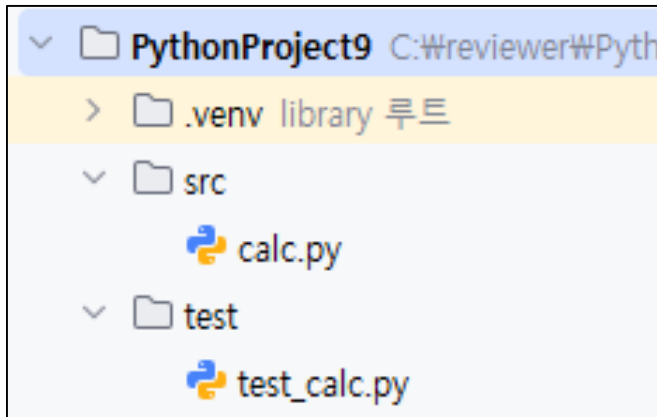
캡처 비활성화

capsys.disabled() 를 이용하면 특정 블록에서 캡처를 끄고 콘솔로 출력한다

```
def test_disabled(capsys):  
    with capsys.disabled():  
        print("이부분은 캡처 되지 않고 출력된다")  
  
    assert capsys.readouterr().out == ""
```

[참고] 소스 코드 경로 처리 -1

test_calc.py 에서는 현재 디렉토리(test) 또는 sys.path 에 등록된 경로만 import 대상으로 인식한다



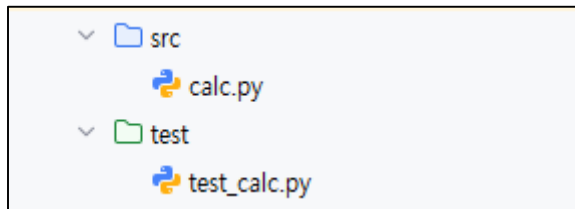
```
calc.py x
1 class Calc:
2     def get_sum(self, a, b):
3         return a + b
```

```
test_calc.py x
1 import calc
2
3 def test_calc():
4     sut = Calc()
5     ret = sut.get_sum(1, 2)
6     assert ret == 3
```

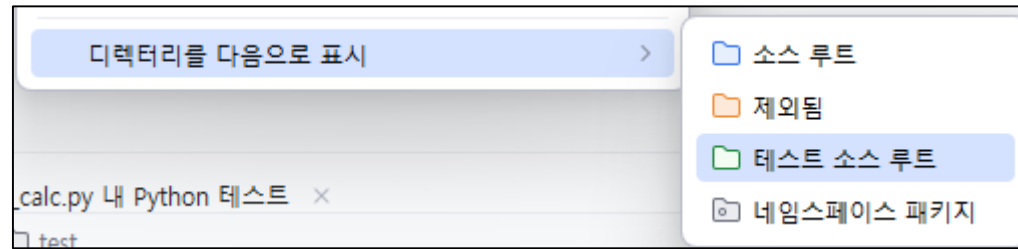
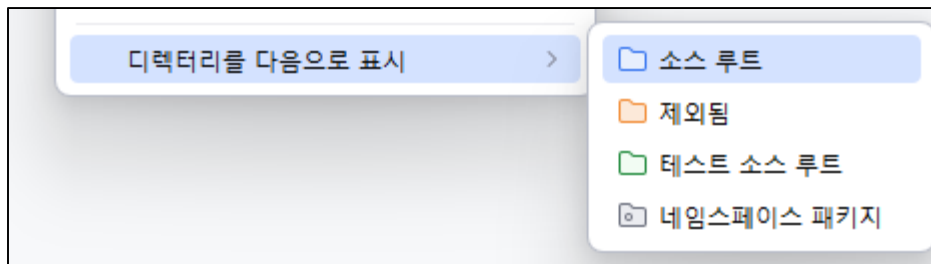
[참고] 소스 코드 경로 처리 -2

src 폴더와 test 폴더의 표시를 변경한다

src 폴더 > 소스 루트 , test 폴더 > 테스트 소스 루트



src 폴더와 test 폴더 우클릭 후 디렉토리 표시를 변경한다



sources root 로 지정된 디렉토리는 내부적으로 sys.path 에 자동 추가된다

* pycharm 에서만 동작하므로 pycharm 외부에서는 설정파일 세팅 필요

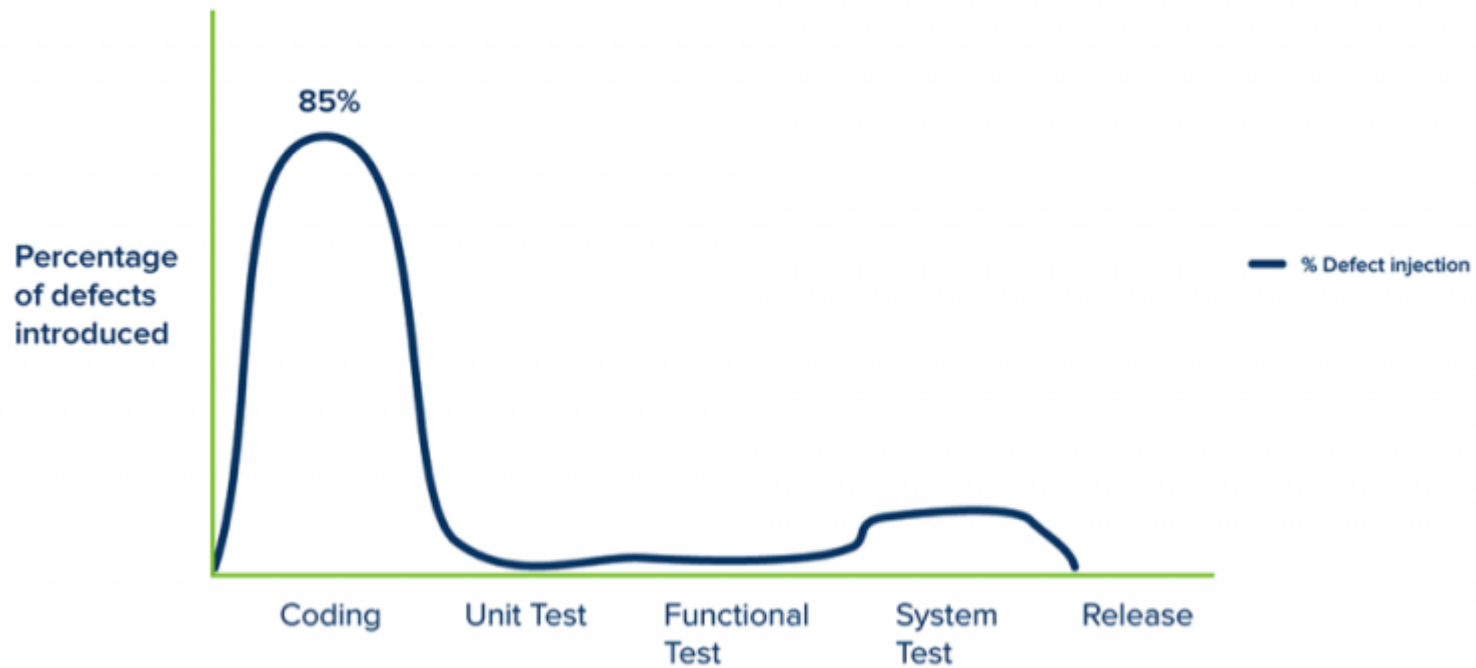
Chapter6

Unit Test 의미

Unit Test 의 의미 : 결함의 비용 최소화

- 결함이 만들어지는 타이밍

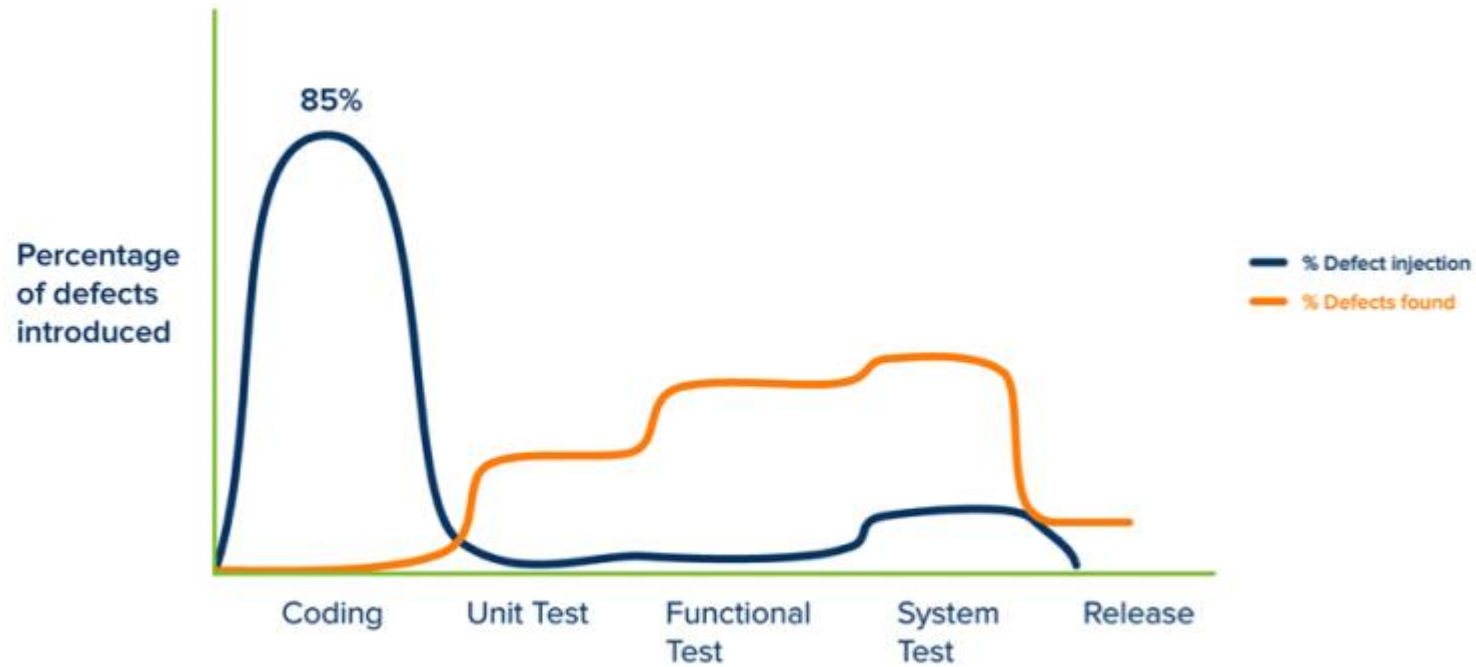
대부분의 버그는 코딩 단계에서 만들어진다.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.

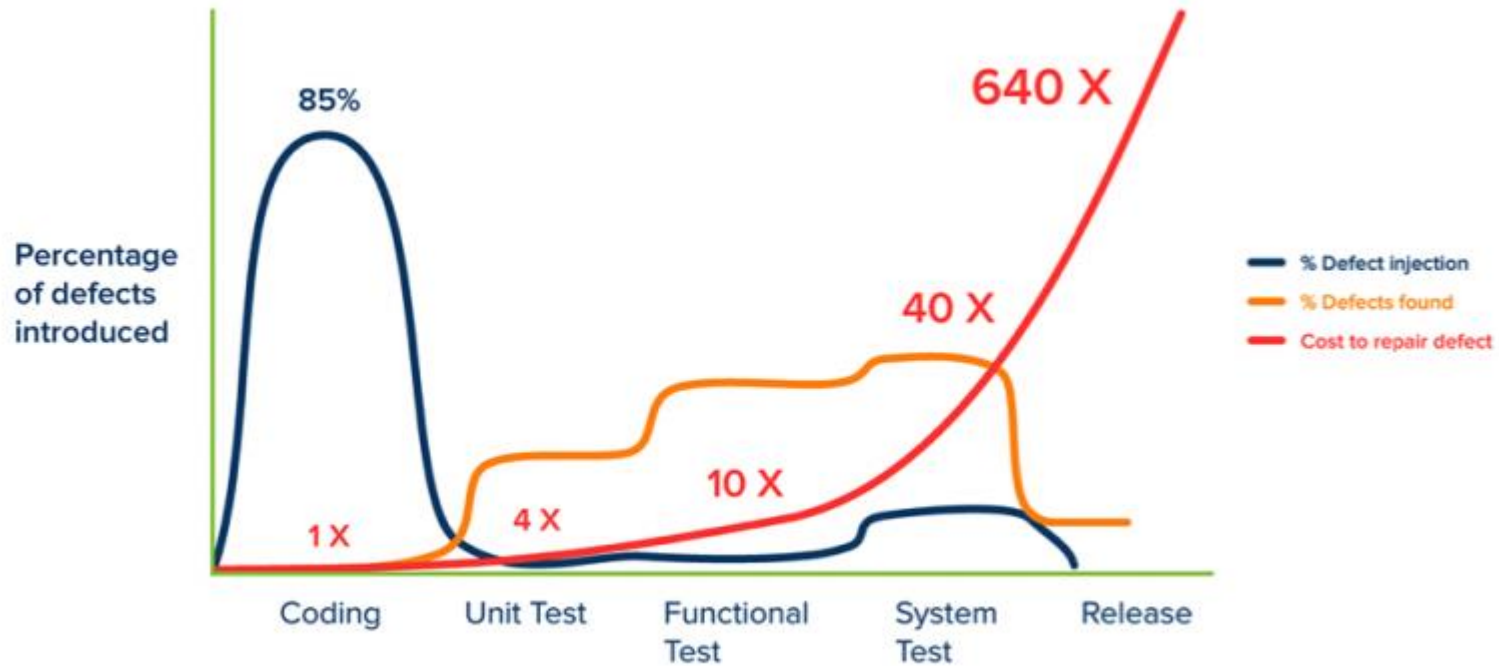
• 결함의 발견

테스트를 시작할 때부터, 많은 결함이 발견된다.



결함 처리 비용

- 개발의 각 단계에서 결함을 수정하는데 드는 비용의 차이
개발이 진행될수록, 결함이 발견될수록 비용이 급격히 증가

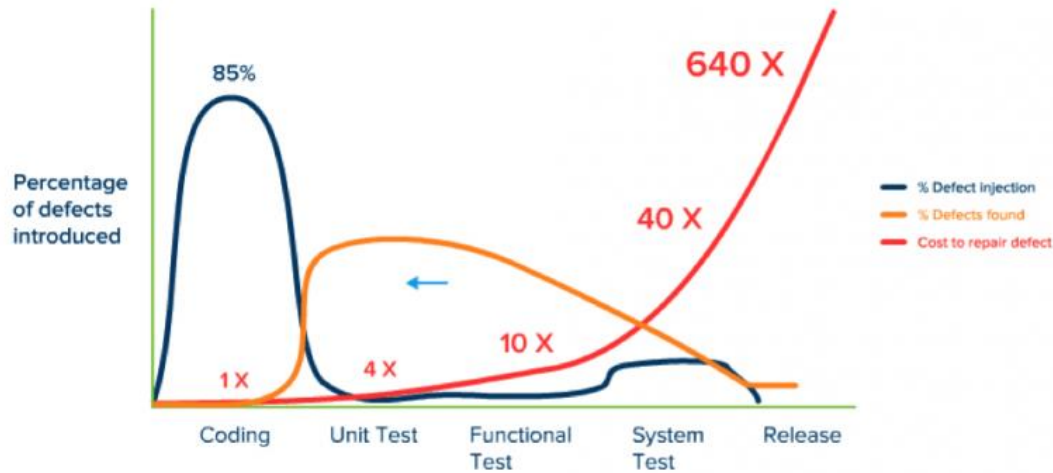


Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.

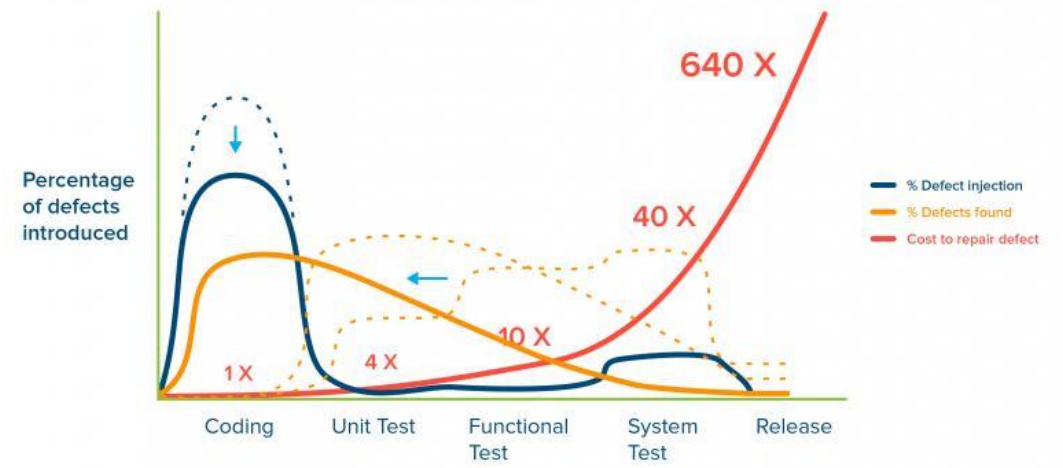
결함 처리 비용

• 테스트를 초기에 수행하면 일어나는 변화

코드의 문제를 일찍 발견할수록 그로 인한 영향이 줄어들고 문제를 해결하는 비용이 줄어든다



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.

[도전] 테스트용 DB를 반환하는 fixture

제공되는 파일 기반 DB 클래스를 테스트하는 코드를 작성한다.

test fixture 를 이용해서 임시디렉토리를 갖춘 DB 인스턴스를 각 테스트별로 제공하도록 한다.

```
class SimpleDB:
    def __init__(self, dir_path: Path):
        self.file = dir_path / "data.txt"
        self.file.touch(exist_ok=True)

    def add(self, line: str):
        with self.file.open("a", encoding="utf-8") as f:
            f.write(line + "\n")

    def get_all(self):
        return self.file.read_text(encoding="utf-8").splitlines()

    def clear(self):
        self.file.write_text("", encoding="utf-8")
```

SimpleDB

+add(str)
+get_all()
+clear()

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/refactoring/DBtest_fixture.py