

파이썬 'pytest'



테스트가 무엇인지 다시 생각해보자

pytest는 unittest에 비해 더 적은 코드로 더 명확한 테스트를 작성할 수 있다

테스트 = 코드가 제대로 작동하는지 확인하는 것

- $1 + 1$ 이 정말 2인지 확인
- 로그인 기능이 제대로 동작하는지 확인
- 계산기가 올바른 답을 내는지 확인



핵심 요약

쉽게 pytest는 내 코드가 잘 동작하는지 자동으로 확인해주는 도구입니다!
손으로 일일이 확인하는 대신 pytest가 대신 확인해줍니다.



pytest란 무엇인가

pytest는 똑똑한 시험 감독관이다

학교에서 시험보고 난 후 감독관이 채점을 한다고 생각해보자

| 내가 직접 채점 (수동 테스트) | 시험 감독관 (pytest) |
|-------------------|-----------------|
| 🤔 답안지 하나하나 확인 | 🤖 자동으로 채점 |
| 😵 실수하기 쉬움 | ✅ 정확하게 확인 |
| ⏰ 시간이 오래 걸림 | ⚡ 빠르게 완료 |
| 📝 매번 다시 확인해야 함 | ⌚ 언제든 다시 실행 가능 |



pytest는 코드를 자동으로 검사해주는 똑똑한 감독관

코드를 바꿀 때마다 "잘 동작하나?"를 자동으로 확인해준다



테스트 없이 코드를 작성한다면 문제가 생긴다

내가 만든 코드를 일일히 확인하기 위해 복잡한 절차를 거쳐야 한다.

```
# 내가 만든 덧셈 함수
def add(a, b):
    return a + b

# 잘 동작하나?
# → print로 확인해봐야 함...
print(add(2, 3)) # 5 나오나?
print(add(-1, 1)) # 0 나오나?
print(add(0, 0)) # 0 나오나?
```

문제점

- 惛 매번 눈으로 결과를 확인해야 함
- 😱 코드가 많아지면 다 확인하기 힘듦
- 🐍 함수의 input, output 이 달라지면서 버그를 놓칠 수 있음



테스트를 사용한다면 어떻게 될까?

장점

- 자동으로 확인한다
- 버튼 한번 딸깍하면 모든 검사를 실행한다
- 기준과 틀린것이 발견되면 그 즉시 자세하게 알려준다

```
def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
```



pytest에 대해서

'pytest' vs 'unittest'

두 도구의 이름을 기준으로 의미를 파헤쳐보자

| 도구 | 이름 의미 | 특징 |
|----------|------------------------|-----------------------|
| unittest | unit(단위) + test(테스트) | Python에 기본 포함된 테스트 도구 |
| pytest | py(Python) + test(테스트) | 따로 설치하는 더 간단한 테스트 도구 |

왜 두 가지나 존재할까?

`unittest` = Python에 기본으로 들어있음 (설치 불필요)

`pytest` = 나중에 만들어진 더 편리한 도구 (설치 필요)



`unittest` 는 "원래 있던 것", `pytest` 는 "더 편하게 만든 것"!

둘 다 테스트를 할 수 있지만, `pytest` 가 더 쉽고 간단하다



pytest에 대해서

'pytest' vs 'unittest'

pytest는 unittest에 비해 더 적은 코드로 더 명확한 테스트를 작성할 수 있다

| 항목 | unittest (내장) | pytest |
|-------|---|----------------------------|
| 문법 | 클래스 기반, <code>self.assertEqual()</code> | 함수 기반, <code>assert</code> |
| 코드량 | 많음 | 적음 |
| 출력 | 기본적 | 상세하고 컬러풀 |
| 플러그인 | 제한적 | 800개 이상 |
| 학습 곡선 | 중간 | 낮음 |



'pytest' vs 'unittest'

앞서 나온 내용을 쉽게 풀어보자면 unittest는 우체국, pytest는 카카오톡이라고 볼 수 있다

| 정식 우편 (unittest) | 카카오톡 (pytest) |
|------------------|---------------|
| 1. 편지지 준비 | 1. 앱 열기 |
| 2. 편지 쓰기 | 2. 메시지 입력 |
| 3. 봉투에 넣기 | 3. 전송! |
| 4. 주소 쓰기 | |
| 5. 우표 붙이기 | |
| 6. 우체통에 넣기 | |

...

핵심요약

둘 다 메시지를 보낼 수 있지만 **카카오톡(pytest)**이 훨씬 간단하다.
테스트도 마찬가지이다!



'pytest' vs 'unittest'

실제 코드로 살펴보자. 둘 모두 똑같은 테스트를 하지만 코드량이 다르다.

unittest 방식 (상대적으로 복잡함)

```
import unittest

class TestMath(unittest.TestCase):
    def test_add(self):
        result = 2 + 3
        self.assertEqual(result, 5)

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

pytest 방식 (상대적으로 간단함)

```
def test_add():
    result = 2 + 3
    assert result == 5
```

→ 총 3줄!

→ 그냥 함수와 assert만 있으면 된다.

→ 총 8줄!

→ class, self, 특별한 메서드 등 복잡하다



pytest 에 대해서

'pytest' vs 'unittest'

unittest에서 pytest로 바꾸면 사라지는 것들

```
# ❌ unittest - 복잡!
import unittest # 1. import 필요

class TestMath(unittest.TestCase): # 2. 클래스 필요
    def test_add(self): # 3. self 필요
        self.assertEqual(2+3, 5) # 4. self.assertEqual 필요

if __name__ == '__main__': # 5. 실행 블록 필요
    unittest.main()
```

```
# ✅ pytest - 간단!
def test_add(): # 그냥 함수!
    assert 2 + 3 == 5 # 그냥 assert!
```



pytest에 대해서

unittest → pytest로 바꾼다면?

unittest의 복잡한 메서드들이 pytest에서는 간단해진다

| unittest (복잡) | pytest (간단) | 의미 |
|--|----------------------------|-----------|
| <code>self.assertEqual(a, b)</code> | <code>assert a == b</code> | a와 b가 같은지 |
| <code>self.assertNotEqual(a, b)</code> | <code>assert a != b</code> | a와 b가 다른지 |
| <code>self.assertTrue(x)</code> | <code>assert x</code> | x가 참인지 |
| <code>self.assertFalse(x)</code> | <code>assert not x</code> | x가 거짓인지 |
| <code>self.assertIn(a, b)</code> | <code>assert a in b</code> | a가 b에 있는지 |



패턴을 발견했나요?

unittest: `self.assertEqual(a, b)`

pytest: `assert a == b`

pytest는 그냥 파이썬 문법 그대로 쓰면 됩니다!



pip란?

pip는 Python 패키지를 설치하는 도구이다.

| 단어 | 의미 |
|-----|--------------------------------------|
| pip | Pip Installs Packages (패키지를 설치하는 도구) |

💡 이해 돋기

pip = 파이썬 앱스토어 같은 것! ⇒ 다른 사람이 만든 프로그램을 설치할 수 있어요.

pytest는 파이썬의 테스트 도구이며 파이썬에 내장되어 있는 unittest와 달리 별도로 설치해야 한다.



쉽게 말해서

pip로 pytest를 설치하면 테스트를 쉽게 할 수 있다.
스마트폰에 앱을 똑딱해서 설치하는 것처럼!



pytest 설치부터 실행까지

pytest 설치하기

pytest 설치 → 앱 설치와 같다

| 스마트폰 | Python |
|------------|-----------------------|
| ■ 앱스토어 열기 | ■ 터미널(명령 프롬프트) 열기 |
| 🔍 앱 검색 | pip install pytest 입력 |
| ⬇ 설치 버튼 클릭 | Enter 누르기 |
| ✓ 설치 완료! | ✓ 설치 완료! |

→ **한 줄 정리**

스마트폰에 앱을 설치하듯이
컴퓨터에 pytest를 설치하면 테스트 기능을 사용할 수 있어요!

pytest에서의 assert



일반 Python의 assert는 메시지가 단순하지만 pytest는 실패 원인을 상세히 보여줍니다!

- 기본 형태
- 동작 원리를 살펴보자

```
assert 조건식  
assert 조건식, "실패 시 메시지"
```

```
# 조건이 True → 아무 일도 안 일어남 (통과)  
assert 1 + 1 == 2
```

```
# 조건이 False → AssertionError 발생 (실패)  
assert 1 + 1 == 3 # AssertionError!
```

```
# 메시지 포함  
assert 1 + 1 == 3, "1+1은 3이 아닙니다"  
# AssertionError: 1+1은 3이 아닙니다
```

pytest 사용을 위해 알아야 할 것들

다양한 assert 표현식

```
# 동등성 비교  
assert result == expected  
assert result != wrong_value
```

```
# 포함 여부  
assert "hello" in "hello world"  
assert 3 in [1, 2, 3, 4, 5]  
assert "error" not in log_message
```

```
# 동일성 (같은 객체인지)  
assert result is None  
assert result is not None
```

```
# 참/거짓  
assert is_valid # True인지  
assert not is_empty # False인지
```

```
# 비교 연산  
assert len(items) > 0  
assert price >= 0  
assert age < 150
```

```
# 타입 확인  
assert isinstance(result, list)  
assert type(result) == dict
```



[간단 실습] pytest 실행해보기

STEP 1 → pytest 설치

앞서 *unittest*을 학습하면서 설정해둔 가상환경을 이용해주세요!

- 1단계: pytest 설치
 - 가상환경 내에서 아래 명령어를 입력해서 pytest를 설치한다.

```
pip install pytest
```

- 2단계: pytest 설치 확인 (버전은 상이할 수 있으나 문제 없음)

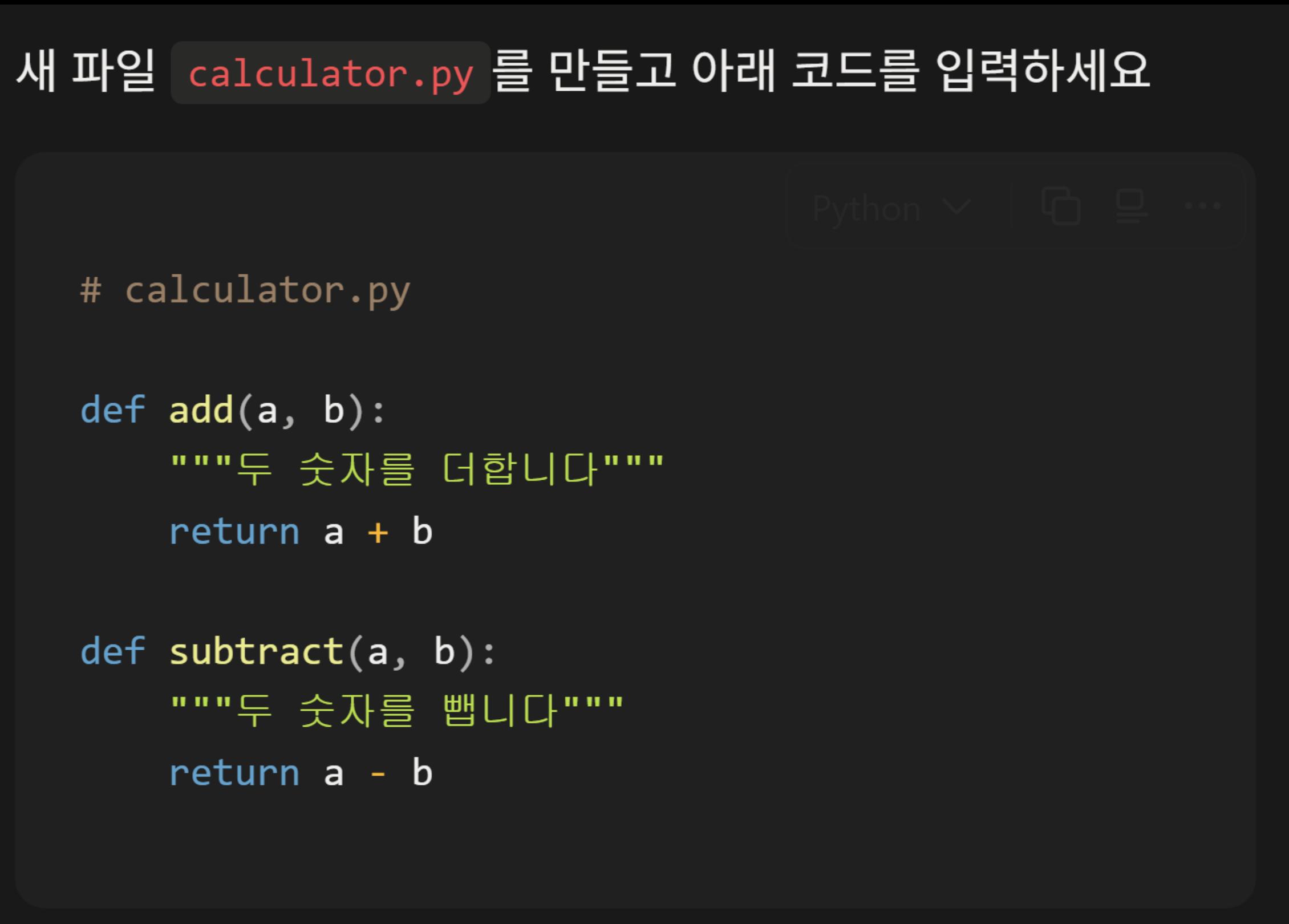
```
$ pytest --version
pytest 8.3.5
```



[간단 실습] pytest 실행해보기

STEP 2 → 테스트 할 코드 만들기

새 파일 `calculator.py` 를 만들고 아래 코드를 입력하세요



```
# calculator.py

def add(a, b):
    """두 숫자를 더합니다"""
    return a + b

def subtract(a, b):
    """두 숫자를 뺍니다"""
    return a - b
```



[간단 실습] pytest 실행해보기

STEP 2 → 테스트 할 코드 만들기

같은 폴더에 `test_calculator.py` 파일을 만드세요

```
# test_calculator.py

from calculator import add, subtract

def test_add():
    """덧셈 테스트"""
    result = add(2, 3)
    assert result == 5

def test_add_negative():
    """음수 덧셈 테스트"""
    result = add(-1, 1)
    assert result == 0

def test_subtract():
    """뺄셈 테스트"""
    result = subtract(10, 3)
    assert result == 7
```



[간단 실습] pytest 실행해보기

STEP 2 → 테스트 할 코드 만들기

폴더 구조는 아래와 같이 될 것이다

```
📁 프로젝트 폴더
  ├── calculator.py      ← 테스트할 코드
  └── test_calculator.py  ← 테스트 코드
```



중요!

두 파일이 같은 폴더에 있어야 합니다!



[간단 실습] pytest 실행해보기

STEP 3 → 테스트 실행

- 터미널에서 파일이 있는 폴더로 이동한 것을 확인한 후 아래 명령어를 입력한다

```
pytest
```

- 실행 결과

```
$ pytest
=====
 test session starts =====
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0
rootdir: ...
configfile: pyproject.toml
collected 3 items

tests\test_calculator.py ... [100%]
```



[간단 실습] pytest 실행해보기

STEP 3 → 테스트 실행

- 더 자세한 결과를 보고 싶으면 '-v' 를 붙이면 된다.

```
pytest -v
```

- 실행 결과

```
$ pytest -v
===== test session starts =====
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0 --
\PythonProject\venv38\Scripts\python.exe
cachedir: .pytest_cache
rootdir: [REDACTED]
configfile: pyproject.toml
collected 3 items

tests/test_calculator.py::test_add PASSED [ 33%]
tests/test_calculator.py::test_add_negative PASSED [ 66%]
tests/test_calculator.py::test_subtract PASSED [100%]

===== 3 passed in 0.01s =====
```



[간단 실습] pytest 실행해보기

STEP 3 → 테스트 실행

- 결과 읽는 법

| 부분 | 의미 |
|------------------------------|-----------|
| test_calculator.py::test_add | 파일::함수 이름 |
| PASSED | 통과! |
| [33%] | 전체 진행률 |

옵션 설명

-v 는 verbose(자세한)의 줄임말이에요.
어떤 테스트가 통과했는지 이름을 보고 싶을 때 사용하세요!

- 더 많은 옵션은 '-h' 명령어를 통해 확인 가능합니다.

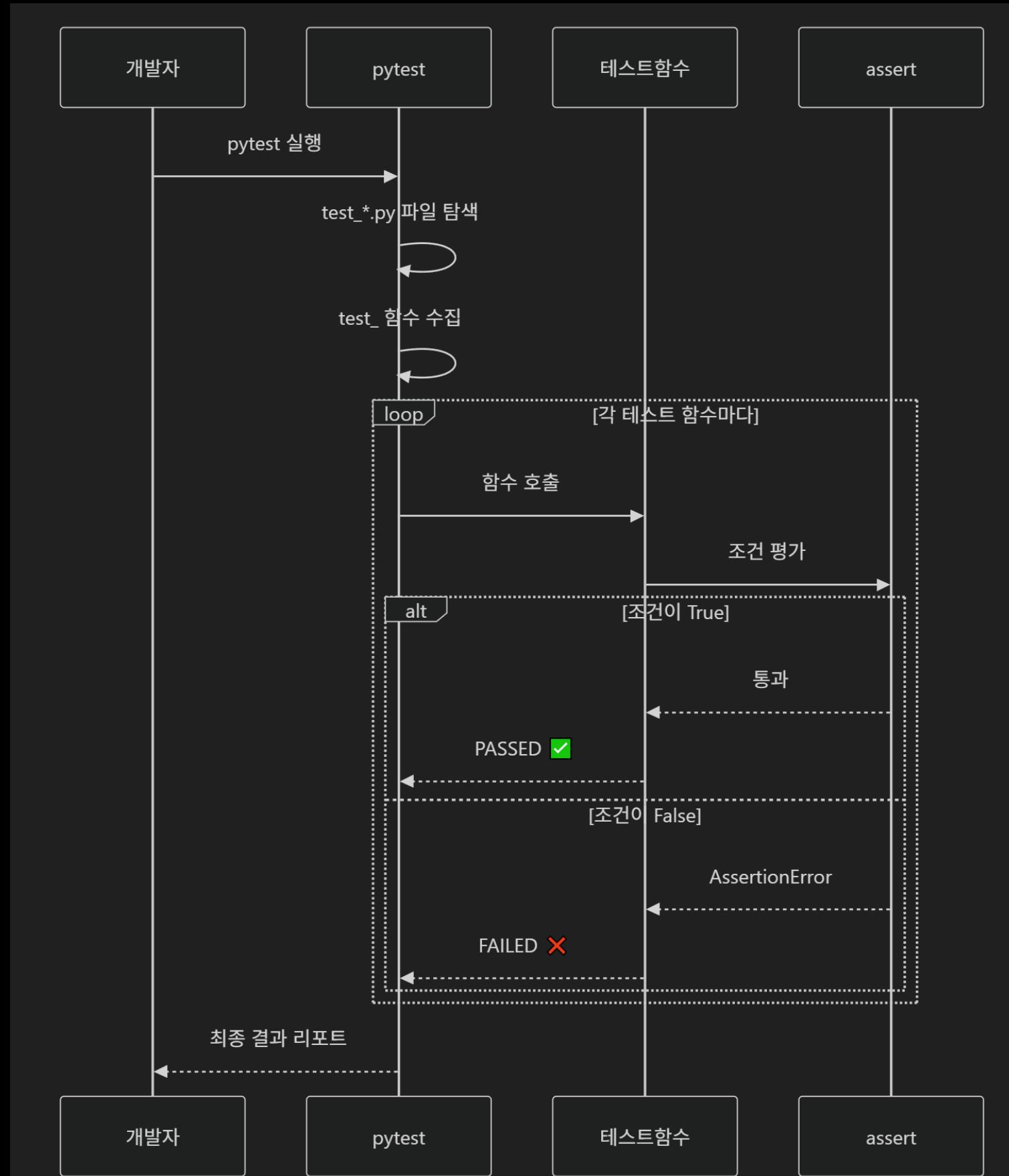
```
$ pytest -h
usage: pytest [options] [file_or_dir] [file_or_dir] [...]

positional arguments:
  file_or_dir

general:
  -k EXPRESSION           Only run tests which match the given substring expression. An
                           expression is a Python evaluable expression where all names are
                           substring-matched against test names and their parent classes.
                           Example: -k 'test_method or test_other' matches all test functions
```

pytest 사용을 위해 알아야 할 것들

pytest 실행 흐름을 살펴보자



- 이미지의 글자가 보이지 않는 경우
 - 이 링크를 확인해주세요
- 링크가 보이지 않으면 아래 링크를 복사하세요
 - <https://kdt-gitlab.elice.io/-/snippets/27>

에러 메세지 읽는 법 (Traceback 해석하기)

```
===== FAILURES =====
____ test_divide ____ ← 실패한 테스트 이름

def test_divide():
    result = divide(10, 3)
>     assert result == 3.33 ← 실패한 줄
E     assert 3.33333333333335 == 3.33 ← 실제값 vs 기대값
E     +  where 3.33333333333335 = divide(10, 3) ← 함수 호출 정보

tests/test_calculator.py:20: AssertionError ← 파일:줄번호
```

에러 메세지 읽는 법 (Traceback 해석하기)

```
===== FAILURES =====
_____  
def test_c  
result  
>     assert  
E     assert  
E     +   wh  
_____
```

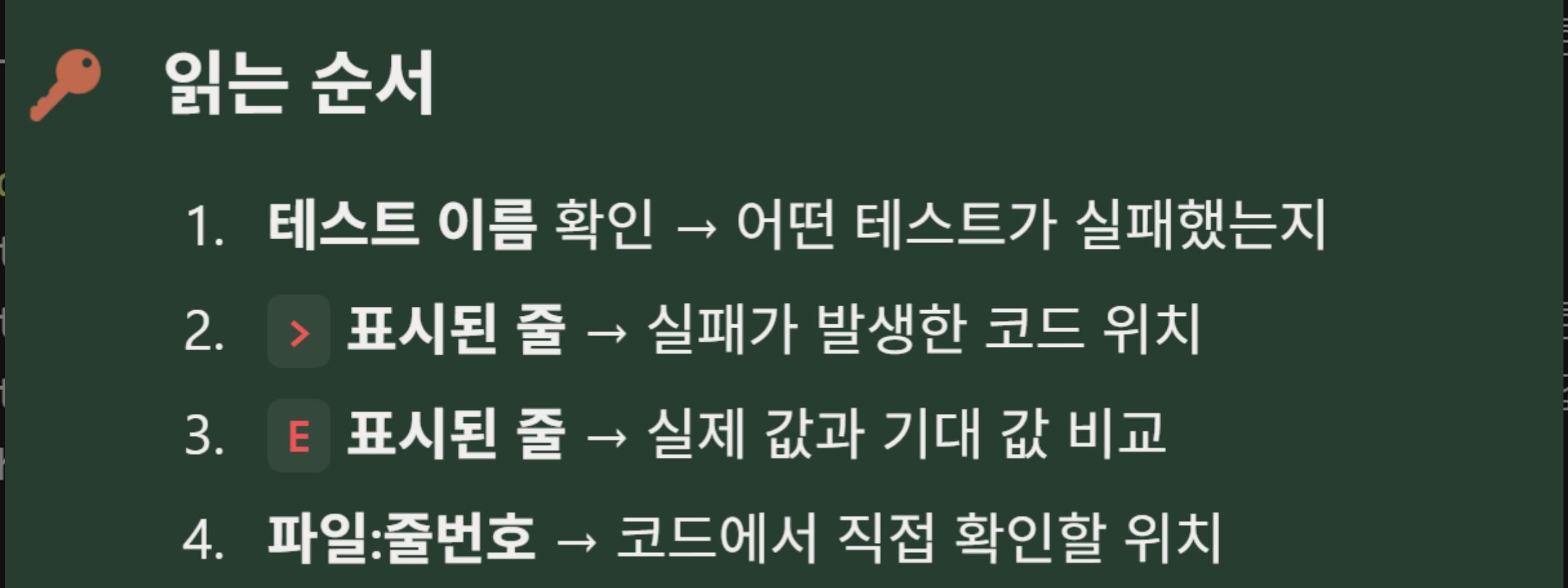
한 테스트 이름

한 줄

값 vs 기대값

호출 정보

← 파일:줄번호



1. 테스트 이름 확인 → 어떤 테스트가 실패했는지
2. > 표시된 줄 → 실패가 발생한 코드 위치
3. E 표시된 줄 → 실제 값과 기대 값 비교
4. 파일:줄번호 → 코드에서 직접 확인할 위치

tests/test_calculator.py:20: AssertionError ← 파일:줄번호

목표: reverse_string 함수를 테스트하는 코드 작성하기

- 주어진 함수 (*src/string_utils.py*)
 - 아래 함수를 직접 타이핑 해서 프로젝트에 생성하자
 - 실행해서 값이 어떻게 나오는지 콘솔에서 확인하자

```
def reverse_string(s):  
    """문자열을 뒤집어 반환합니다."""  
    return s[::-1]
```

목표: reverse_string 함수를 테스트하는 코드 작성하기

- 요구사항

1. `tests/test_string_utils.py` 파일을 생성하세요
2. 다음 케이스를 테스트하는 함수를 각각 작성하세요:
 - 일반 문자열: `"hello"` → `"olleh"`
 - 빈 문자열: `""` → `""`
 - 한 글자: `"a"` → `"a"`
 - 회문(팰린드롬): `"level"` → `"level"`
 - 공백 포함: `"hello world"` → `"dlrow olleh"`
3. pytest로 실행하여 모든 테스트가 통과하는지 확인하세요

목표: reverse_string 함수를 테스트하는 코드 작성하기

실습 답안 안내

코드가 매우 긴 관계로 이 [링크](#)를 통해 코드를 확인해주세요!

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/28> 복사해주세요.



Arrange! Act! Assert!

- AAA 약어가 의미하는 단어를 하나씩 파헤쳐보자

| 단어 | 의미 | 테스트에서 역할 |
|---------|------------|---------------|
| Arrange | 준비하다, 정렬하다 | 테스트에 필요한 것 준비 |
| Act | 행동하다, 실행하다 | 테스트할 코드 실행 |
| Assert | 단언하다, 주장하다 | 결과가 맞는지 확인 |

- 테스트에서의 의미

모든 테스트는 3단계로 이루어집니다

- 1 **Arrange** (준비) → 필요한 것 준비하기
- 2 **Act** (실행) → 테스트할 코드 실행하기
- 3 **Assert** (검증) → 결과 확인하기

쉽게 말해서!

AAA는 테스트를 쓰는 레시피입니다!
"준비 → 실행 → 확인" 이 순서대로 쓰면 됩니다.



테스트 작성 패턴, AAA

Arrange! Act! Assert!

- 요리하는 상황으로 대입해서 생각해보자
- 좀 더 자세히 풀어서 생각해본다면 아래와 같다

| 구분 | 요리 | 테스트 |
|---------|---------|---------------|
| Arrange | ❶ 재료 준비 | 테스트에 필요한 값 준비 |
| Act | ❷ 조리하기 | 함수 실행 |
| Assert | ❸ 맛 확인 | 결과 확인 |

핵심요약

레시피를 따르면 누구나 같은 요리를 만들 수 있듯이
AAA 패턴을 따르면 누구나 읽기 쉬운 테스트를 만들 수 있습니다!

| 라면 끓이기 | 덧셈 테스트 |
|--|---|
| 1. 재료 준비 (Arrange) - 물 500ml - 라면 1개 - 스프 1봉지 | 1. 숫자 준비 (Arrange) - a = 2 - b = 3 - 기대값 = 5 |
| 2. 조리 (Act) - 물 끓이고 - 면과 스프 넣기 | 2. 함수 실행 (Act) - result = add(a, b) |
| 3. 맛 확인 (Assert) - 맛있는가? ✓ | 3. 결과 확인 (Assert) - result == 5? ✓ |



Arrange! Act! Assert!

- AAA를 실제로 살펴보자

```
def add(a, b):  
    return a + b  
  
def test_add():  
    # Arrange (준비)  
    a = 2  
    b = 3  
    expected = 5  
  
    # Act (실행)  
    result = add(a, b)  
  
    # Assert (검증)  
    assert result == expected
```

- 각 단계 설명

```
# Arrange (준비) - 테스트에 필요한 값을 미리 준비  
a = 2  
b = 3  
expected = 5 # 예상하는 결과값  
  
# Act (실행) - 테스트하고 싶은 함수를 실행  
result = add(a, b) # add 함수가 제대로 동작하는지 테스트!  
  
# Assert (검증) - 결과가 예상과 같은지 확인  
assert result == expected # result가 5인지 확인!
```

💡 중요한 팁

주석(# Arrange, # Act, # Assert)을 쓰세요!

그러면 코드를 읽는 사람도, 작성하는 사람도 각 부분이 뭘 하는지 바로 알 수 있습니다!



간단한 테스트는 더 짧게 써도 된다

- 테스트가 간단하면 주석 생략 가능
- 언제 주석을 쓸까?

```
# 간단한 테스트는 이렇게!
def test_add_simple():
    result = add(2, 3)
    assert result == 5

# 더 간단하게 한 줄로!
def test_add_oneline():
    assert add(2, 3) == 5
```

| 테스트 복잡도 | 주석 사용 |
|------------|----------|
| 간단 (1-2줄) | 주석 없이 OK |
| 보통 (3-5줄) | 선택 사항 |
| 복잡 (6줄 이상) | 주석 권장 |



기억하세요: 주석이 있든 없든! 테스트는 항상 준비 → 실행 → 확인 순서입니다!



테스트 작성 패턴, Given-When-Then

Given! When! Then!



Given-When-Then은 테스트를 이야기처럼 쓰는 방법입니다!
"~가 있을 때, ~를 하면, ~해야 한다" 형식으로 씁니다.

- 각각의 단어를 한번 살펴보자

| 단어 | 의미 | 테스트에서 역할 |
|-------|-----------|-----------|
| Given | ~가 주어졌을 때 | 테스트 전제 조건 |
| When | ~를 하면 | 테스트할 행동 |
| Then | ~해야 한다 | 예상 결과 |



테스트 작성 패턴, Given-When-Then

Given! When! Then!

- 영화 시나리오를 쓰는 상황에 대입해보자

| 구분 | 영화 시나리오 | 테스트 시나리오 |
|-------|----------------------|---------------------|
| Given | [상황 설정] 주인공이 배고프다 | [전제 조건] 계산기가 있다 |
| When | [사건 발생] 주인공이 라면을 끓인다 | [행동] $2 + 3$ 을 계산한다 |
| Then | [결과] 주인공이 배부르다 | [예상 결과] 결과가 5이다 |



핵심 정리

영화 시나리오처럼 "상황 → 사건 → 결과"로 테스트를 쓰면
누구나 쉽게 이해할 수 있는 테스트가 됩니다!



테스트 작성 패턴, Given-When-Then

기본 예시 코드 → 성적 평균 테스트

- 평균 성적을 구하는 프로그램의 테스트 코드를 Given-When-Then 패턴을 적용하면 아래와 같다.
- 주석을 문장처럼 쓰면서 개발하는 걸 권장한다. 그래야 테스트가 무엇을 하는지 바로 알 수 있다.

```
def calculate_average(scores):
    return sum(scores) / len(scores)

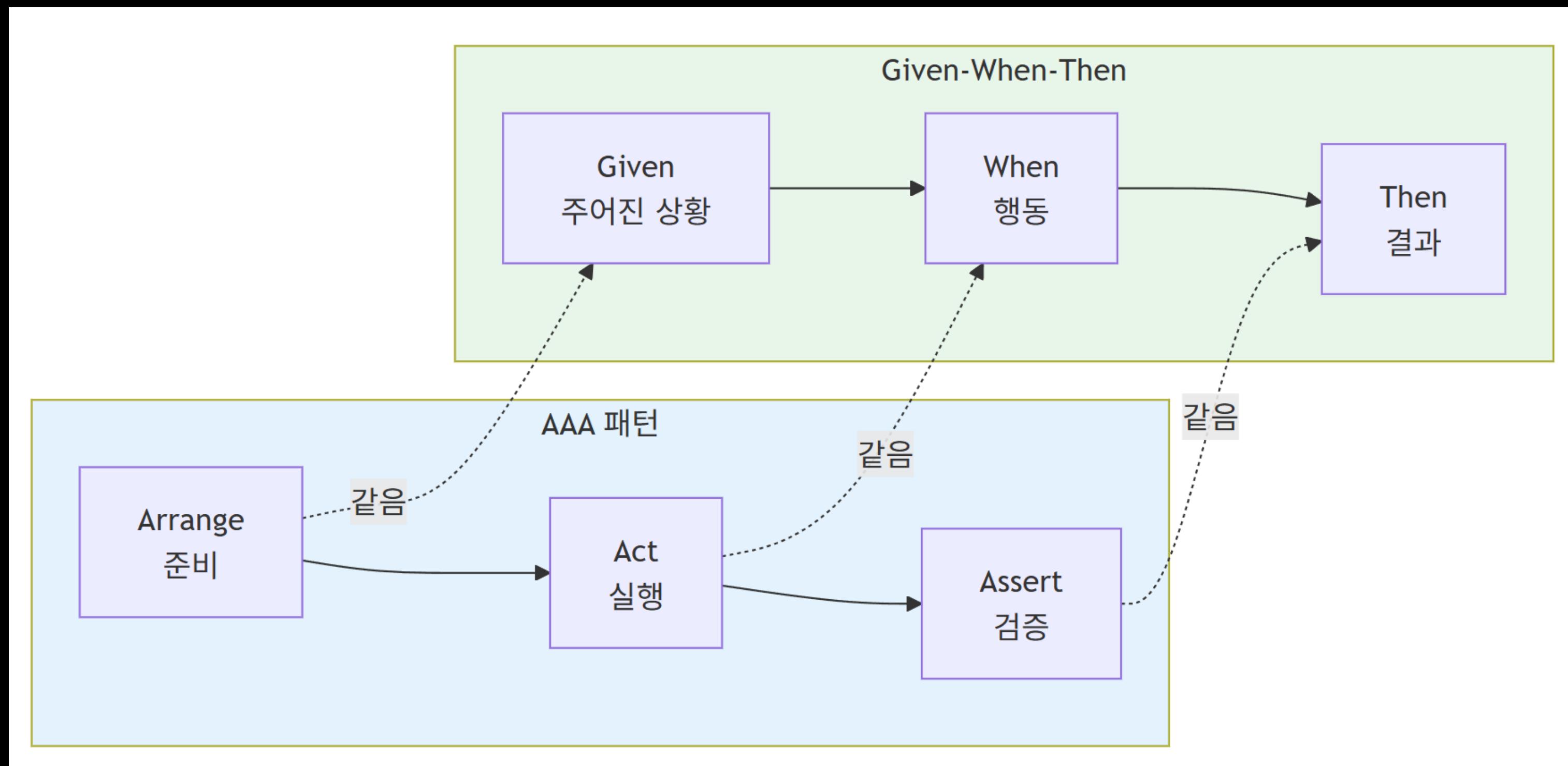
def test_average():
    # Given: 점수 리스트가 있을 때
    scores = [80, 90, 100]

    # When: 평균을 계산하면
    result = calculate_average(scores)

    # Then: 결과는 90이어야 한다
    assert result == 90
```

AAA, Given-When-Then 은 같은 구조

- 두 패턴은 이름만 다르고 구조는 같습니다.



AAA, Given-When-Then 은 같은 구조

- 두 패턴은 본질적으로 같은 구조를 가진다. (준비 → 실행 → 검증)
- 하지만 사용 맥락 및 함께 일하는 대상이 누구냐에 따라 선택이 바뀐다.
 - AAA → 개발자 간 코드리뷰가 주요 커뮤니케이션일 때
 - GWT → 기획, QA, 개발 간 공통 언어가 필요 할 때 (== BDD Framework 사용 시)
 - 테스트 문서를 비개발자가 읽을 수 있어야 할 때

| 기준 | AAA | Given-When-Then |
|--------|------------------|------------------|
| 주요 독자 | 개발자 | 개발자 + 비개발자 |
| 테스트 유형 | 단위 테스트 | 인수/통합 테스트 |
| 작성 방식 | 코드 중심 | 자연어 중심 |
| 도구 | unittest, pytest | Cucumber, Behave |
| 초점 | 구현 | 행위/시나리오 |



[실습] 두 패턴으로 테스트 작성

목표: AAA, Given-When-Then 모두를 사용해서 테스트 작성하기

- 아래 함수를 AAA, GWT 패턴으로 각각 테스트하세요!
- 테스트 대상 코드 및 요구사항

```
def get_discount_price(price, discount_percent):  
    """할인된 가격을 계산합니다"""  
    discount = price * discount_percent / 100  
    return price - discount
```

요구사항

1. `test_discount_aaa` : AAA 패턴으로 테스트
 - 10000원 상품, 20% 할인 → 8000원 확인
2. `test_discount_gwt` : Given-When-Then 패턴으로 테스트
 - 5000원 상품, 10% 할인 → 4500원 확인

목표: AAA, Given-When-Then 모두를 사용해서 테스트 작성하기

실습 답안 안내

코드가 매우 긴 관계로 이 [링크](#)를 통해 코드를 확인해주세요!

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/29> 복사해주세요.



Fixture 이해하기

Fixture의 단어를 파헤쳐보자

- 영어로 "고정된 것", "설비"라는 뜻을 가지고 있다.

| 단어 | 의미 |
|-------|------|
| fix | 고정하다 |
| -ture | ~한 것 |

- Fixture = "고정된 것" → 테스트 할 때 **항상 준비되어 있는 것!**



쉽게 말해서..

Fixture는 테스트 전에 미리 준비해두는 것입니다!

요리하기 전에 재료를 미리 준비해두는 것처럼 말이지요.



Fixture 가 없을 때 문제 상황을 살펴보자

- 테스트 코드에 반복이 생긴다. → 유지보수성이 떨어진다!

```
def test_더하기():
    calculator = Calculator() # 매번 만들어야 함
    result = calculator.add(2, 3)
    assert result == 5

def test_빼기():
    calculator = Calculator() # 또 만들어야 함!
    result = calculator.subtract(5, 3)
    assert result == 2

def test_곱하기():
    calculator = Calculator() # 또또 만들어야 함!!
    result = calculator.multiply(4, 5)
    assert result == 20
```



문제 발생!

`Calculator()` 를 만드는 코드가 3번이나 반복됩니다!
테스트가 100개면 100번 반복해야 할까요?



Fixture를 사용하면 한 번만 정의하면됩니다.
그리고 여러 테스트에서 재사용할 수 있습니다!



Fixture 이해하기

학교에서 급식을 제공할 때와 제공하지 않을 때를 생각해봅시다.

- 급식이 있을 때 vs 없을 때
- 테스트에 적용한다면?

| 급식이 없다면? 😞 | 급식이 있다면? 😊 |
|----------------|----------------------|
| 학생 1: 도시락 싸오기 | 급식실에서 한 번에 준비 |
| 학생 2: 도시락 싸오기 | 학생들은 받아서 먹기만 하면 됨 |
| 학생 3: 도시락 싸오기 | 반복 작업 없음! |
| 모두가 각자 준비해야 함! | 준비는 급식실이 담당! |

급식 없이 (`fixture` 없이):
`test_1: Calculator` 만들기 → 테스트
`test_2: Calculator` 만들기 → 테스트 (반복!)
`test_3: Calculator` 만들기 → 테스트 (또 반복!)

급식 있으면 (`fixture` 사용):
`fixture: Calculator` 미리 준비
`test_1: Calculator` 받아서 → 테스트
`test_2: Calculator` 받아서 → 테스트 (준비 코드 없음!)
`test_3: Calculator` 받아서 → 테스트 (준비 코드 없음!)



Fixture = 급식실

테스트에 필요한 것을 미리 준비해두면 각 테스트는 받아서 사용만 하면 됩니다!



Fixture 이해하기

이제 첫 번째 Fixture 를 만들어보자

Step 1: `@pytest.fixture` 붙이기

```
import pytest

@pytest.fixture      # ← 이 한 줄이 핵심!
def calculator():
    return Calculator()
```



잠깐! `@`는 뭔가요?

`@pytest.fixture`에서 `@`는 데코레이터입니다.

"이 함수를 fixture로 만들어줘!"라는 의미예요.

지금은 "함수 위에 `@` 붙이면 fixture가 된다"고만 이해하면 됩니다!



Fixture 이해하기

이제 첫 번째 Fixture 를 만들어보자

Step 2: 테스트에서 사용하기

```
def test_더하기(calculator):    # ← 인자로 받기만 하면 됨!
    result = calculator.add(2, 3)
    assert result == 5

def test_빼기(calculator):        # ← 같은 이름으로 받으면 자동 연결
    result = calculator.subtract(5, 3)
    assert result == 2
```

⚠️ **Fixture 이름 = 함수 인자 이름**

@pytest.fixture로 만든 함수 이름이 calculator이면
테스트 함수의 인자 이름도 calculator로 똑같이 적어야 합니다!
pytest가 이름이 같은 것을 찾아서 자동으로 연결해줍니다.



이제 첫 번째 Fixture 를 만들어보자

- 전체 코드

```
import pytest

# ===== 테스트 대상 =====
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b
```

- 코드가 긴 관계로 이 [링크](#)를 통해 확인해주세요
- 링크가 보이지 않으면 아래 링크를 복사하세요
 - <https://kdt-gitlab.elice.io/-/snippets/30>

```
# ===== Fixture 정의 =====
@pytest.fixture
def calculator():
    """Calculator 객체를 미리 만들어둡니다"""
    pass
```



이제 첫 번째 Fixture 를 만들어보자

- 실행 결과

```
$ pytest -v -s

Calculator 생성!          # test_더하기 전에 생성
test_더하기 PASSED

Calculator 생성!          # test_빼기 전에 또 생성 (매번 새로!)
test_빼기 PASSED

test_일반 PASSED          # fixture 안 쓴

===== 3 passed ======
```



포인트

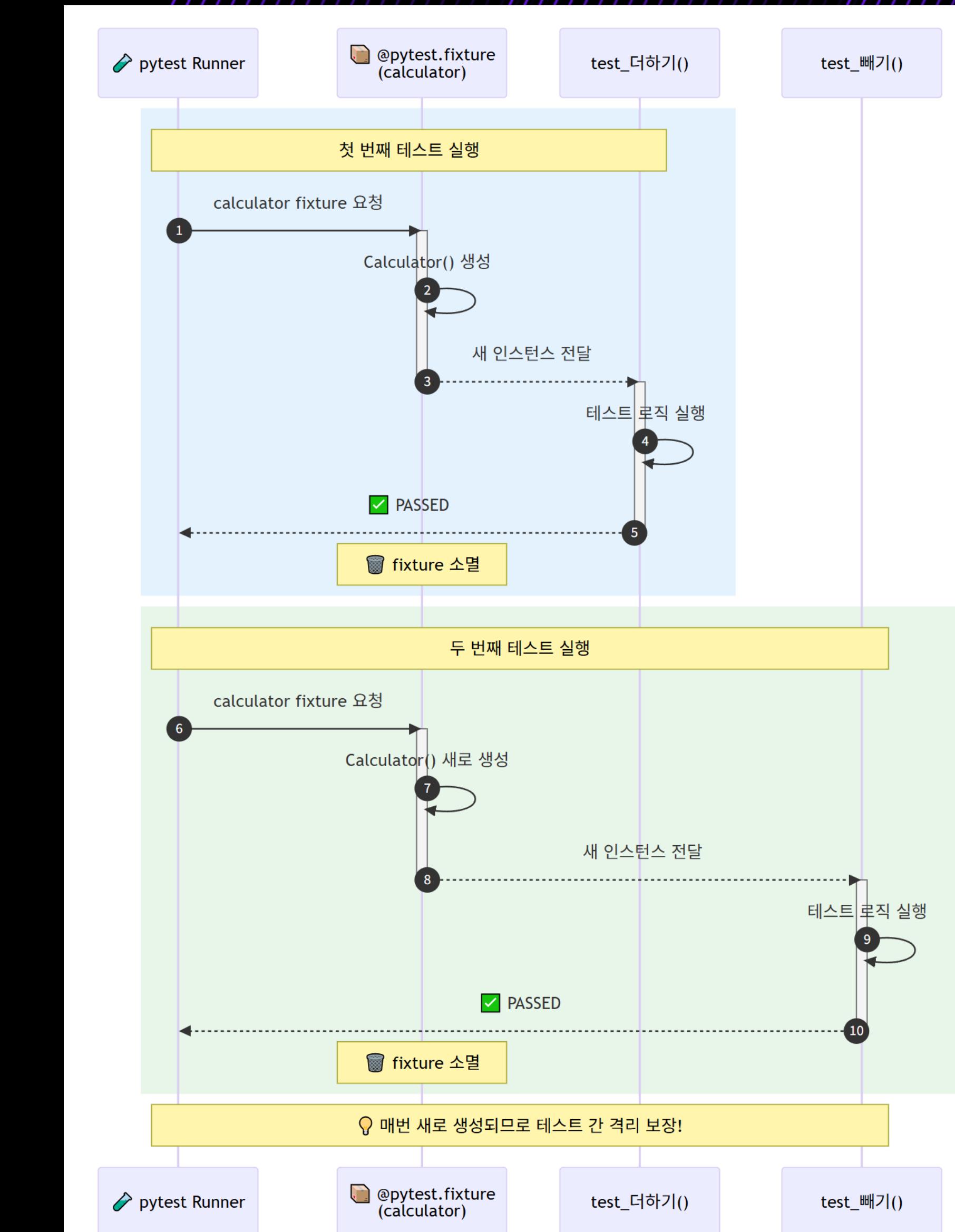
각 테스트마다 **새로운** Calculator가 만들어집니다!
그래서 테스트끼리 서로 영향을 주지 않아요.



Fixture 이해하기

Fixture의 생명주기

- 이미지의 글자가 보이지 않는 경우
 - 이 [링크](#)를 확인해주세요
 - 링크가 보이지 않으면 아래 링크를 복사하세요
 - <https://kdt-gitlab.elice.io/-/snippets/31>





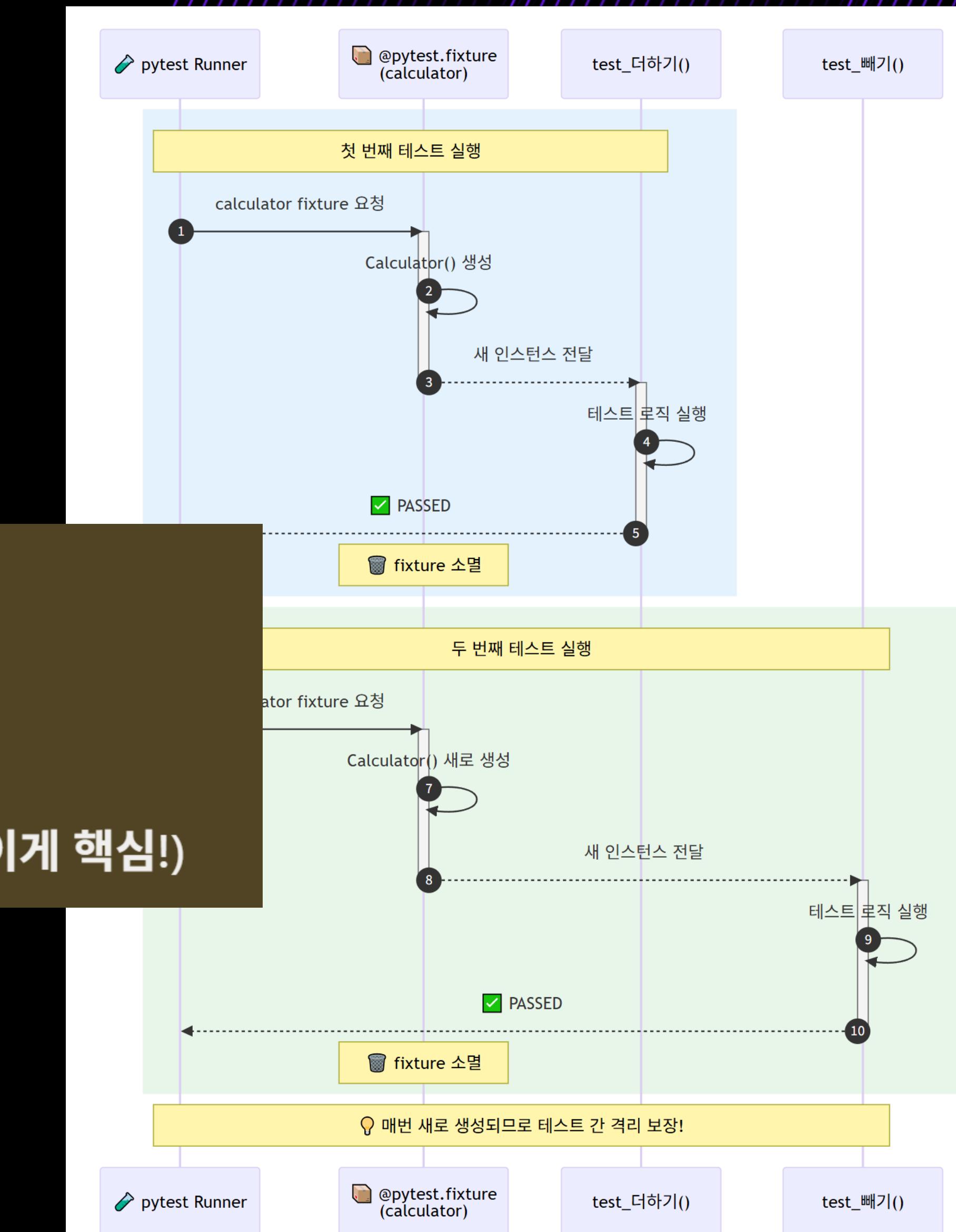
Fixture 이해하기

Fixture의 생명주기



왜 매번 새로 만들까요?

test_1에서 Calculator를 변경해도
test_2는 새로운 Calculator를 받으니까
테스트끼리 서로 영향을 주지 않습니다! (이게 핵심!)





Fixture 이해하기

Fixture는 여러 개 만들 수 있다

- 여러개의 fixture를 만들 수 있다.
- 테스트에서는 필요한 것만 골라쓸 수 있다

```
import pytest

@pytest.fixture
def 사과():
    return {"이름": "사과", "가격": 1000}

@pytest.fixture
def 바나나():
    return {"이름": "바나나", "가격": 1500}

@pytest.fixture
def 빈_장바구니():
    return []
```

```
def test_사과_가격(사과):
    """fixture 1개 사용"""
    assert 사과["가격"] == 1000

def test_과일_비교(사과, 바나나):
    """fixture 2개 사용"""
    assert 바나나["가격"] > 사과["가격"]

def test_장바구니에_담기(빈_장바구니, 사과):
    """fixture 2개 사용"""
    빈_장바구니.append(사과)
    assert len(빈_장바구니) == 1
```



[실습] 학생 성적 관리 테스트

목표 : Student 클래스를 테스트하기 위한 fixture 를 만들자

- 테스트 대상 코드

```
class Student:  
    def __init__(self, name):  
        self.name = name  
        self.scores = []  
  
    def add_score(self, score):  
        self.scores.append(score)  
  
    def average(self):  
        if len(self.scores) == 0:  
            return 0  
        total = sum(self.scores)  
        return total / len(self.scores)
```

요구사항

1. **student** fixture를 만드세요 - Student("철수")를 반환
2. 테스트 3개를 작성하세요:
 - **test_학생_이름** : 이름이 "철수"인지 확인
 - **test_점수_추가** : 점수 80 추가 후 scores에 80이 있는지 확인
 - **test_평균_계산** : 점수 80, 90, 100 추가 후 평균이 90인지 확인

목표 : Student 클래스를 테스트하기 위한 fixture 를 만들자

실습 답안 안내

코드가 매우 긴 관계로 이 [링크](#)를 통해 코드를 확인해주세요!

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/32> 복사해주세요.

[실습] 문자열 뒤집기 함수 테스트 작성

목표 : Student 클래스를 테스트하기 위한 fixture 를 만들자

- 실행 결과

```
$ pytest -v  
  
test_학생_이름 PASSED  
test_점수_추가 PASSED  
test_평균_계산 PASSED  
  
3 passed
```

fixture 이름과 테스트 인자 이름이 같아야 합니다!

@pytest.fixture 아래의 함수 이름: student

테스트 함수의 인자 이름: student

⇒ pytest가 이름을 보고 자동으로 연결해줍니다!

✓ 핵심 포인트

- @pytest.fixture 로 준비 코드를 한 번만 작성
- 테스트 함수 인자에 같은 이름을 적으면 자동 연결
- 각 테스트마다 새로운 Student가 생성됨 (서로 영향 없음)



[심화] Fixture 이해하기

Scope, yield 를 사용한 고급제어 방법



아래 내용은 fixture에 익숙해진 나중에 학습하세요!

- Scope → fixture 유지 범위 조절

```
@pytest.fixture(scope="module") # 파일당 1번만 생성
def config():
    return {"timeout": 30}
```

- yield → 테스트 후 정리 작업

```
@pytest.fixture
def temp_file():
    # 준비
    f = open("test.txt", "w")
    yield f          # 테스트에 전달
    # 정리 (테스트 끝난 후)
    f.close()
```

이 개념들은 파이썬의 decorator, yield 문법을
익히시고 난 이후에 공부해주세요!



Parameterize에 대해서

Parameterize의 단어를 파헤쳐보자

- 이 단어는 두 단어가 합쳐진 것이다

| 단어 | 의미 |
|-----------|-----------------|
| parameter | 매개변수 (함수에 넣는 값) |
| -ize | ~화하다 (만들다) |

☞ parameterize는 하나의 테스트로 여러 값을 검사하는 기능입니다!
같은 테스트를 여러 번 복사할 필요가 없어요.

- pytest에서의 의미

```
# 하나의 테스트 함수로 여러 값을 검사!
@pytest.mark.parametrize("입력값", [값1, 값2, 값3])
def test_something(입력값):
    ...
```



Parameterize에 대해서

복사기에 빗대어서 이해해보자

- 시험지를 만들 때 복사기를 사용한다고 생각해보자

방법 1: 하나씩 손으로 쓰기 (비효율적!)

시험지 1: $2 + 3 = ?$

시험지 2: $5 + 7 = ?$

시험지 3: $10 + 20 = ?$

→ 매번 새로 써야 함! 😞

방법 2: 복사기 사용 (효율적!)

시험지 틀: $\underline{\quad} + \underline{\quad} = ?$

숫자만 바꿔서 대량 생산! 📄📄📄

→ 틀은 하나, 숫자만 다르게! 😊



핵심: 테스트 함수는 **하나만** 만들고, **값만 여러 개** 넣으면 됩니다!



Parameterize에 대해서

왜 Parameterize가 필요할까?

- 문제 상황 → 비슷한 테스트가 계속 반복된다!
- Parameterize를 사용해서 해결!

```
# ❌ 복사-붙여넣기의 반복...
def test_grade_95():
    assert get_grade(95) == 'A'

def test_grade_85():
    assert get_grade(85) == 'B'

def test_grade_75():
    assert get_grade(75) == 'C'

def test_grade_65():
    assert get_grade(65) == 'D'

def test_grade_55():
    assert get_grade(55) == 'F'

# 5개의 함수가 거의 똑같아요! 😞
```

```
# ✅ parametrize로 한 번에!
@pytest.mark.parametrize("score, expected", [
    (95, 'A'),
    (85, 'B'),
    (75, 'C'),
    (65, 'D'),
    (55, 'F'),
])
def test_grade(score, expected):
    assert get_grade(score) == expected

# 함수 1개로 5가지 테스트! 😊
```



Parameterize에 대해서

@pytest.mark.parametrize 기본 문법

- 기본 구조
- 문법 설명

```
import pytest

@pytest.mark.parametrize("변수이름", [값1, 값2, 값3])
def test_함수이름(변수이름):
    # 테스트 코드
    pass
```

```
@pytest.mark.parametrize("number", [10, 30, 90])
#                                     ↑          ↑
#                               변수 이름     테스트할 값들 (리스트)

def test_positive(number):
#                           ↑
#                   위의 변수 이름과 같아야 함!
    assert number > 0
```



잠깐! @는 뭔가요?

@pytest.mark.parametrize에서 @는 데코레이터입니다.

"이 함수에 특별한 기능을 추가해줘!"라는 의미예요.

지금은 "함수 위에 @ 붙이면 parametrize 기능이 추가된다"고만 이해하면 됩니다!



@pytest.mark.parametrize 기본 문법

- 실행 결과

```
===== test session starts =====
collecting ... collected 3 items
tests/test_calculator.py::test_positive[10] PASSED [ 33%]
tests/test_calculator.py::test_positive[30] PASSED [ 66%]
tests/test_calculator.py::test_positive[90] PASSED [100%]
=====
3 passed in 0.01s =====
```

값이 바뀌는 것을 볼 수 있다.

- 앞서 코드에서 테스트 할 값들로 설정 한 10, 30, 90 이 자동으로 바뀌면서 테스트 된다.
- 이게 없다면 반복문을 사용해서 모든 자원을 직접 핸들링 해야 하는 엄청난 불편함이 생긴다.

[실습] Parameterize 사용해보기

목표 : 양수인지 확인하기

- 직접 손으로 치는게 핵심입니다!
- 프로젝트에 아래 코드를 작성해주세요

```
import pytest

@pytest.mark.parametrize("number", [1, 5, 10, 100])
def test_is_positive(number):
    """양수인지 확인하는 테스트"""
    assert number > 0
```

[실습] Parameterize 사용해보기

목표 : 양수인지 확인하기

- 실행 결과

```
$ pytest -v
=====
test session starts =====
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0 --
\venv38\Scripts\python.exe
cachedir: .pytest_cache
rootdir: 
configfile: pyproject.toml
collected 4 items

tests/test_calculator.py::test_is_positive[1] PASSED [ 25%]
tests/test_calculator.py::test_is_positive[5] PASSED [ 50%]
tests/test_calculator.py::test_is_positive[10] PASSED [ 75%]
tests/test_calculator.py::test_is_positive[100] PASSED [100%]

===== 4 passed in 0.01s =====
```

[1], [5] 와 같이 설정한대로 값이 바뀌는 것을 볼 수 있다.



[실습] Parameterize 사용해보기

목표 : 값 여러 개씩 테스트

- 입력값과 예상 결과를 함께 테스트해보자
- 값이 여러 개 필요 할 때는 튜플로 묶어서 사용 할 수 있다.
- 직접 손으로 치고 실행 결과를 눈으로 확인하자

```
import pytest

def add(a, b):
    """두 숫자를 더합니다"""
    return a + b

@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),          # 1 + 2 = 3
    (0, 0, 0),          # 0 + 0 = 0
    (5, 5, 10),         # 5 + 5 = 10
    (10, 20, 30),       # 10 + 20 = 30
])
def test_add(a, b, expected):
    """덧셈 테스트"""
    result = add(a, b)
    assert result == expected
```



[실습] Parameterize 사용해보기

목표 : 값 여러 개씩 테스트

- 실행 결과

```
$ pytest -v
=====
 test session starts =====
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0 --
\venv38\Scripts\python.exe
cachedir: .pytest_cache
rootdir: 
configfile: pyproject.toml
collected 4 items

tests/test_calculator.py::test_add[1-2-3] PASSED [ 25%]
tests/test_calculator.py::test_add[0-0-0] PASSED [ 50%]
tests/test_calculator.py::test_add[5-5-10] PASSED [ 75%]
tests/test_calculator.py::test_add[10-20-30] PASSED [100%]

=====
 4 passed in 0.01s =====
```

마찬가지로 값이 설정한 대로 바뀌는 것을 볼 수 있다.

[실습] Parameterize 사용해보기

목표 : 값 여러 개씩 테스트

- 문법 설명

```
@pytest.mark.parametrize("a, b, expected", [  
    #           ↑  ↑      ↑  
    #           변수 3개 (콤마로 구분)  
  
    (1, 2, 3),    # a=1, b=2, expected=3  
    (0, 0, 0),    # a=0, b=0, expected=0  
])  
  
def test_add(a, b, expected):  
    #           ↑  ↑      ↑  
    #           위의 변수 이름과 같아야 함!
```