

# 파이썬 유닛테스트 (Python 'unittest')



# 테스트란 무엇인가?



## 자동차 출고 전 검사

브레이크가 제대로 작동하는가?

에어백이 정상인가?

엔진 오일 수준은 적절한가?



## 요리사의 맛보기

간이 맞는가?

익힘 정도가 적절한가?

플레이팅이 괜찮은가?



## 비행기 이륙 전 점검

연료량 확인

계기판 정상 작동 확인

날개 상태 확인



# 테스트란 무엇인가?

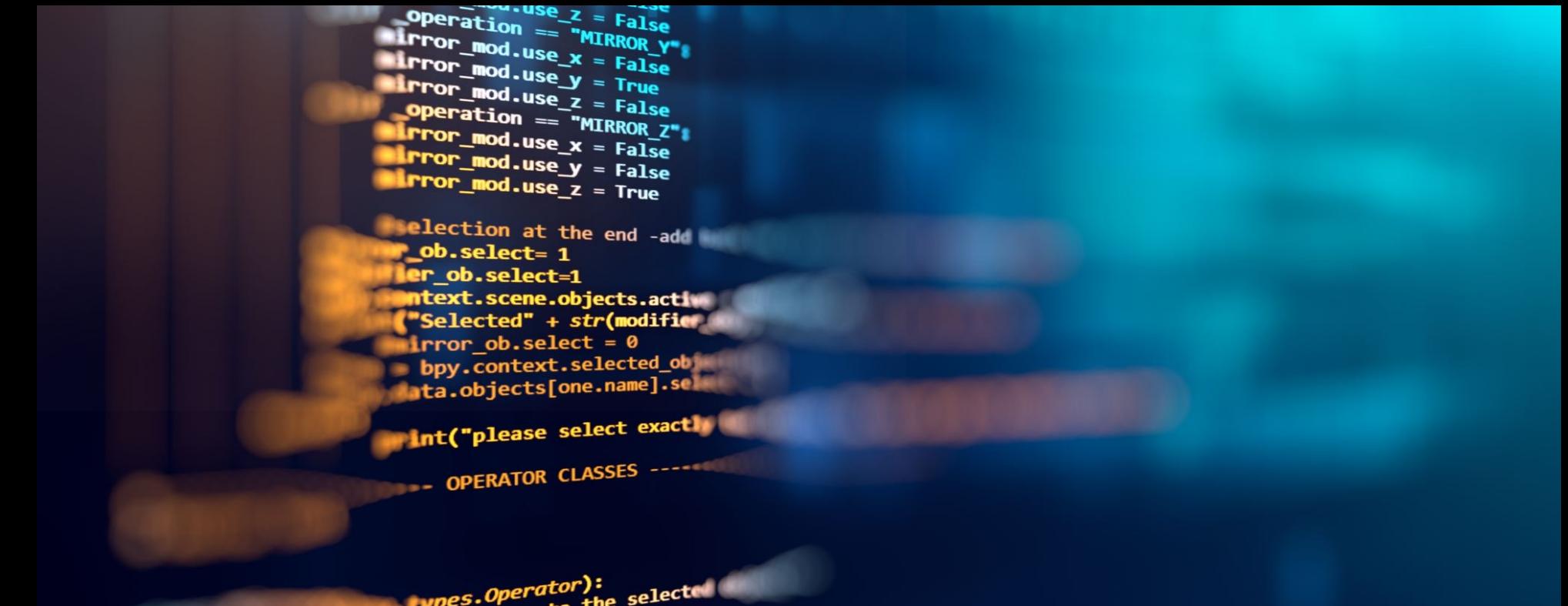
**테스트 없는 개발 = 안전벨트 없이 운전하는 것**

**테스트는 "만약을 위한 보험"이 아니라, 개발의 필수 안전장치입니다.**



## 안전벨트 없이 운전

- 평소에는 문제없이 운전 가능
- 사고 발생 시 치명적 피해
- "지금까지 괜찮았으니까"라는 착각
- 사고 후 후회해도 되돌릴 수 없음



## 테스트 없이 개발

- 평소에는 문제없이 동작
- 버그 발생 시 서비스 장애
- "지금까지 잘 돌아갔으니까"라는 착각
- 장애 후 복구해도 신뢰 손상



# 왜 테스트가 필요한가?

- IBM 연구 결과
  - 버그는 개발 중에 발견 하는것이 비용 효율적이다.
  - 운영 환경에서 발견된 버그를 수정하는 비용은 개발 단계에서 발견했을 때보다 100배 이상 비싸다.

발견 시점	상대 비용	설명
개발 중 (Unit Test)	1x	코드 작성 직후 발견, 즉시 수정
통합 테스트	10x	여러 모듈 연결 후 발견
QA 테스트	50x	전체 시스템 테스트 중 발견
운영 환경 (Production)	100x+	실제 사용자가 버그 경험



# Unit Test란 무엇인가

## Unit Test의 어원



### Unit (단위)

라틴어 *unus* (하나)에서 유래

→ 더 이상 나눌 수 없는 최소 단위

프로그래밍에서는 함수, 메서드, 클래스 등



### Test (시험)

라틴어 *testum* (도가니)에서 유래

→ 금속의 순도를 검증하는 도구

프로그래밍에서는 코드의 정확성 검증



Unit Test = "가장 작은 코드 단위를 도가니에 넣어 검증한다"



# Unit Test란 무엇인가

## 레고블록 품질검사와 Unit Test를 비교해보자

### 레고 품질검사

- 빨간 2x4 블록 → 색상 확인 ✓
- 결합부 → 끼움 강도 테스트 ✓
- 크기 → 규격 측정 ✓
- 표면 → 스크래치 검사 ✓

### Unit Test

- `add()` 함수 → 덧셈 결과 확인 ✓
- `validate()` → 입력값 검증 ✓
- `format()` → 출력 형식 확인 ✓
- `calculate()` → 계산 정확도 ✓



**핵심:** 개별 블록이 모두 정상이어야, 완성된 세트도 정상입니다.



## 자동차 공장을 통해 테스트의 종류를 살펴보자



자동차 한 대는 약 30,000개의 **부품**으로 구성되어 있다.  
만약 조립 후에만 테스트한다면?

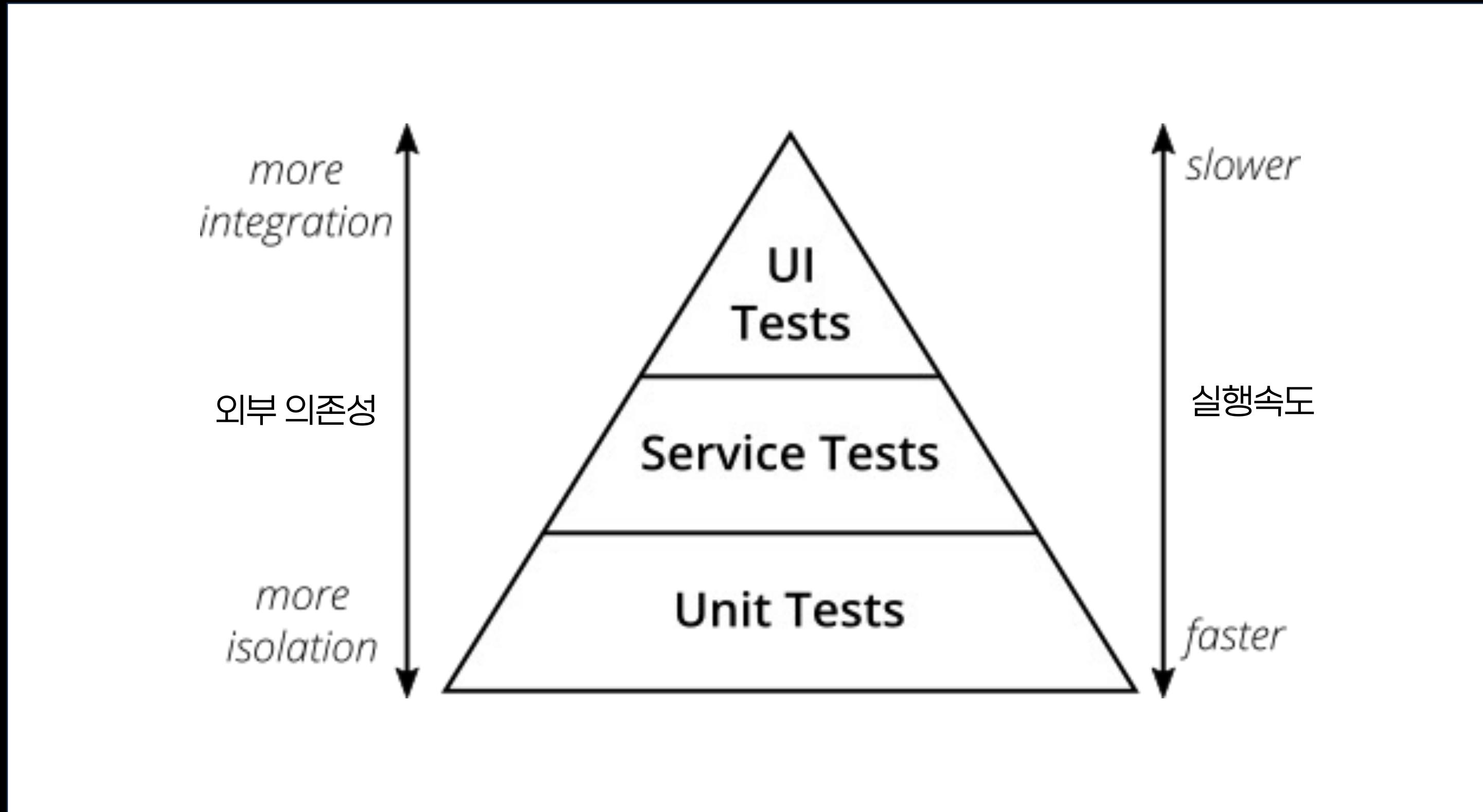
검수 방식	문제점	테스트 비유
완성 후 검수	불량 부품 찾기 위해 전체 분해 필요	E2E 테스트만 수행
부품별 검수	불량 즉시 발견, 교체 용이	Unit Test 수행

### 실제 자동차 공장 프로세스

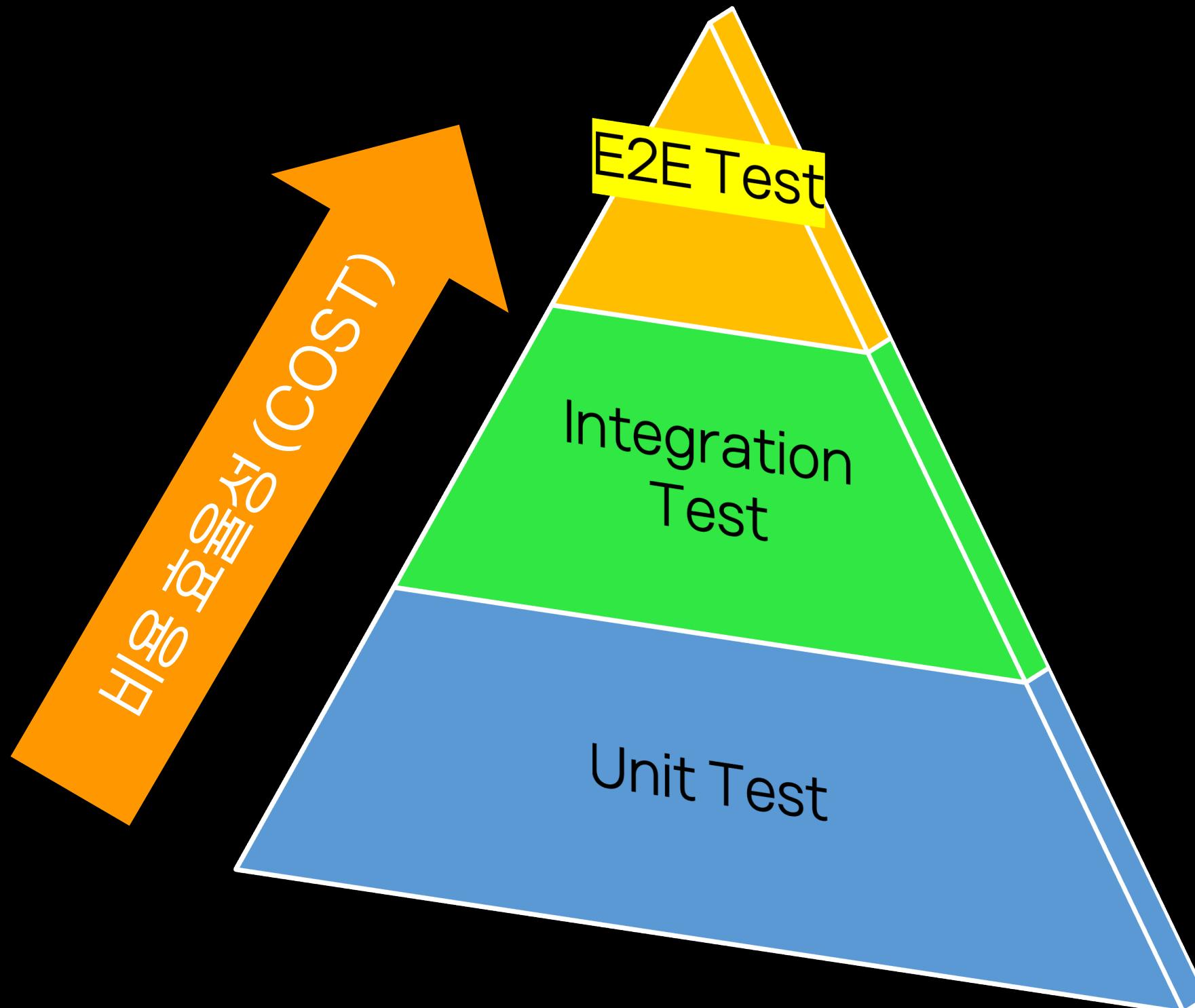
1. **볼트 하나** → 강도 테스트 (Unit Test)
2. **엔진 조립체** → 출력 테스트 (Integration Test)
3. **완성차** → 주행 테스트 (E2E Test)

# ☰ 테스트 피라미드 (Test Pyramid)

마틴 파울러, 마이크 콘, 켄트백, 구글 등은 테스트 피라미드를 수용하고 있다.



# 테스트 피라미드 (Test Pyramid)



유형	Flaky 확률	원인
Unit	낮음	외부 의존성 없음
Integration	중간	DB, API 상태 의존
E2E	높음	네트워크, 타이밍, 렌더링 등

## \* Flaky Test

- 동일코드인데 실행할 때마다 성공, 실패가 달라지는 테스트

# 테스트 피라미드 (Test Pyramid)

## 근거가 되는 연구 / 사례

Google Testing Blog에서 발표한 내용

- 70/20/10 비율을 따르는 프로젝트가 가장 높은 개발자 생산성과 낮은 결함 탈출률을 보였다

Martin Fowler의 분석

- E2E 테스트는 신뢰도는 높지만, 느리고 취약(brittle)하다.
- 피라미드의 꼭대기에 최소한으로 유지해야 한다.

Kent Beck (TDD 창시자)

- 단위 테스트는 설계 도구다. 많을수록 코드 품질이 올라간다.

●  
⋮  
●

<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>

<https://martinfowler.com/bliki/TestPyramid.html>

<https://wiki.c2.com/?UnitTestingIsDesign>

# ☰ 테스트 피라미드 (Test Pyramid)

## 테스트 피라미드를 왜 따라야 할까?

### 빠른 피드백 (Fast Feedback)

- 피라미드의 하위 레벨(Unit) 테스트를 통해 버그를 조기에 발견합니다.

### 비용 효율성 (ROI)

- 상위 레벨로 갈수록 유지보수 비용과 실행 시간이 기하급수적으로 증가합니다.

### 안정성 확보

- 기반이 튼튼한(Unit Test가 많은) 프로젝트일수록 리팩토링과 기능 추가가 안전합니다.

테스트 유형	속도	비용	검증 범위
Unit Test	⚡ 매우 빠름	💰 낮음	함수/메서드 단위
Integration Test	🏃 보통	💰💰 중간	모듈 간 상호작용
E2E Test	🐢 느림	💰💰💰 높음	전체 시스템



# Unit Test vs 다른 테스트 비교

## 테스트 유형별 상세 비교



### Unit Test 강점

빠른 피드백으로 문제를 조기에 발견하고 정확한 위치를 즉시 파악할 수 있습니다.

구분	Unit Test	Integration Test	E2E Test
테스트 대상	함수, 메서드	모듈 간 연결	전체 시스템
실행 시간	밀리초	초 ~ 분	분 ~ 시간
외부 의존성	없음 (Mock 사용)	일부 있음	전체 포함
실패 시 원인	즉시 파악 가능	범위 좁혀야 함	파악 어려움
작성 주체	개발자	개발자/QA	QA
실행 빈도	매 커밋마다	매일/PR마다	릴리즈 전

## 유닛테스트의 F.I.R.S.T 원칙을 알아보자

- ❖ 로버트 C. 마틴의 저서 클린코드 (Clean Code)에서 소개된 유명한 개념이다.

원칙	의미	무엇을 (What)	왜 (Why)	어떻게 (How)
Fast	빠르게	수천 개 테스트를 수 초 내 실행	느리면 개발자가 안 돌림	외부 의존성(DB, 네트워크) 제거
Independent	독립적	테스트 간 순서·상태 공유 없음	실패 원인 즉시 특정	setUp/tearDown으로 격리
Repeatable	반복 가능	언제, 어디서 실행해도 동일 결과	Flaky 테스트 방지	랜덤값·시간·환경 의존 제거
Self-validating	자가 검증	성공/실패를 코드가 판단	수동 확인 제거	assertion으로 명시적 검증
Timely	적시에	프로덕션 코드와 함께 작성	테스트 가능한 설계 유도	TDD 또는 코드 직후 작성



# 첫 번째 Unit Test 작성하기

Step 1: 테스트 대상 함수

```
calculator.py  ×  
1 # calculator.py  
2 def add(a, b):  
3     """두 숫자를 더합니다."  
4     return a + b  
5
```



## 첫 번째 Unit Test 작성하기

Step 2: 테스트 코드

```
py test_calculator.py ×  
1 # test_calculator.py  
2 import unittest  
3 from calculator import add  
4  
5  
6 ▶ class TestCalculator(unittest.TestCase):  
7  
8 ▶     def test_add_positive_numbers(self):  
9         """양수 덧셈 테스트"""  
10        result = add(a: 2, b: 3)  
11        self.assertEqual(result, second: 5) # 2 + 3 = 5 인지 확인  
12  
13 ▶     def test_add_negative_numbers(self):  
14         """음수 덧셈 테스트"""  
15        result = add(-1, -1)  
16        self.assertEqual(result, -2) # -1 + -1 = -2 인지 확인  
17  
18  
19 ▶     if __name__ == '__main__':  
20         unittest.main()
```



## 첫 번째 Unit Test 작성하기

### Step 3: 실행 하는 방법

목적 (Purpose)	명령어 예시 (Command)	상세 설명
전체 실행	python -m unittest discover	현재 위치와 하위 폴더의 모든 테스트 파일(test_*.py)을 자동으로 찾아 실행합니다.
상세 결과 보기	python -m unittest -v	(추천) 테스트 함수명과 통과 여부(OK/FAIL)를 한 줄씩 자세히 보여줍니다. (Verbose를 의미한다.)
특정 파일 실행	python -m unittest tests/test_login.py	전체를 돌리지 않고, 특정 파일 하나만 실행합니다.
특정 클래스 실행	python -m unittest tests.test_login.Login	특정 파일 내의 특정 테스트 클래스 하나만 실행합니다.
특정 함수 실행	python -m unittest tests.test_login.Login.test_user	가장 작은 단위인 테스트 함수(메서드) 하나만 콕 집어 실행합니다.
빠른 중단	python -m unittest -f	테스트 실행 중 첫 번째 에러가 발생하면 즉시 중단합니다. (Failfast)
키워드 검색	python -m unittest -k "user"	테스트 함수나 클래스 이름에 "user"가 포함된 것들만 골라서 실행합니다.



## 첫 번째 Unit Test 작성하기

Step 4: 실행 결과

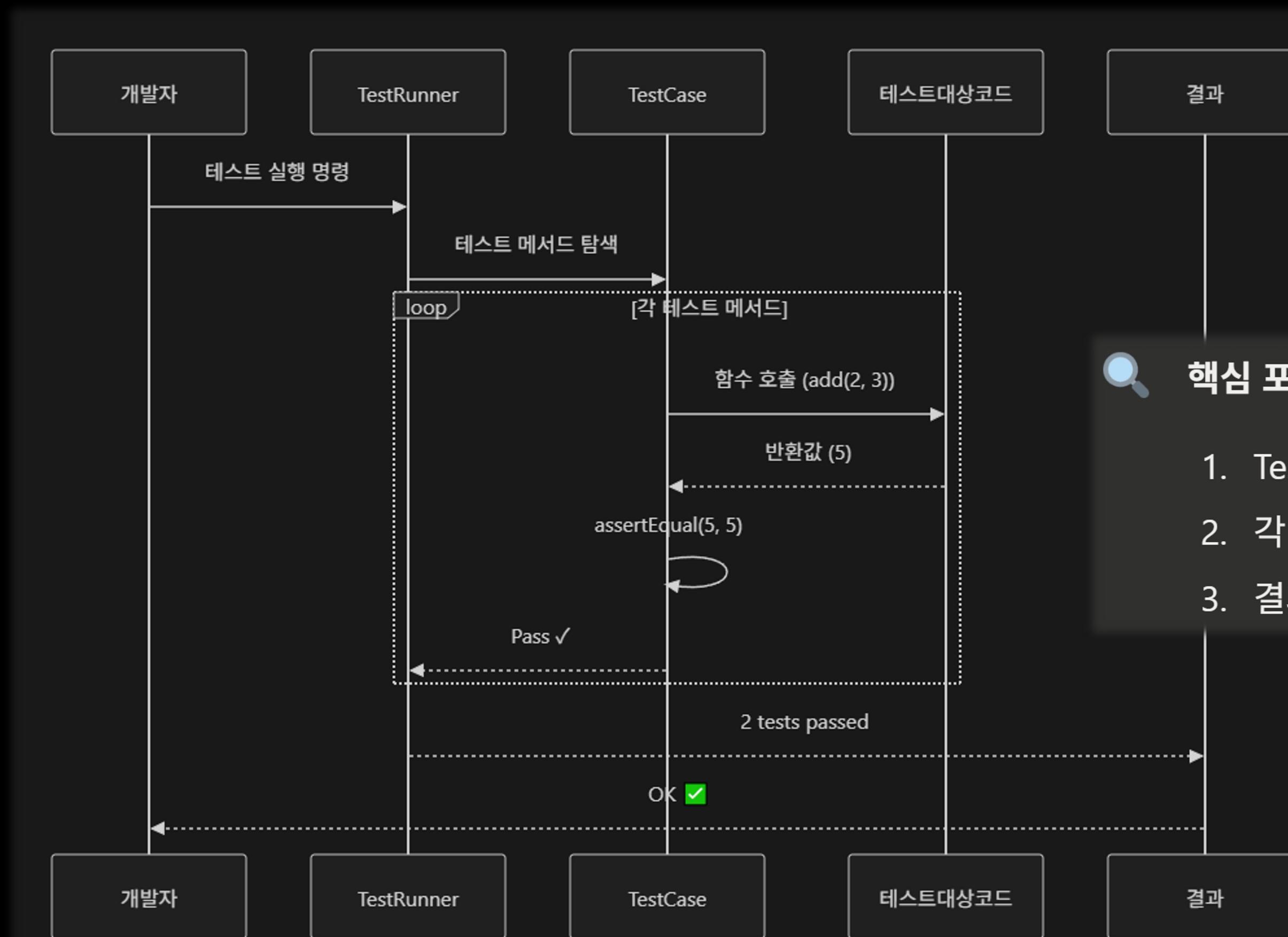
The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says '실행' and 'test\_calculator.TestCalculator의 Python 테스트'. The toolbar has icons for run, stop, and refresh. A dropdown menu is open, showing '테스트 결과' (Test Results) which is selected, and '7ms'. Below it, a green checkmark indicates '2 테스트 통과' (2 tests passed) out of '총 2개의 테스트' (2 total tests), completed in '7ms'. The main pane displays the terminal output of the test run:

```
Testing started at 오전 2:58 ...
Launching unittests with arguments python -m u
Ran 2 tests in 0.007s
OK
종료 코드 0(으)로 완료된 프로세스
```



## [실습 후 분석] 가장 간단한 테스트

# Unit Test 실행 흐름을 살펴보자



### 핵심 포인트:

1. TestRunner가 `test_`로 시작하는 메서드를 자동 탐색
2. 각 테스트는 독립적으로 실행
3. 결과는 자동으로 집계되어 보고

## 문제 1. 덧셈 함수가 Unit Test 대상인 이유 설명하기

### 문제

아래 `add()` 함수가 Unit Test의 좋은 대상인 이유를 F.I.R.S.T 원칙에 따라 설명하세요.

원칙	적용 여부	이유
Fast	O / X	(작성)
Isolated	O / X	(작성)
Repeatable	O / X	(작성)
Self-validating	O / X	(작성)
Timely	O / X	(작성)

```
calculator.py ×
# calculator.py
def add(a, b):
    """두 숫자를 더합니다."""
    return a + b
```

**문제 2. 이 함수를 테스트 할 때 어떤 입력값을 테스트 해야 할까요?**

→ 어떤 분석법을 사용해야 하는지 생각해보세요

**문제 3**

**테스트 케이스는 최소 몇 개가 필요할까요?**

**테스트 케이스의 이름을 모두 적어주세요.**

→ 문제 2에서 선택한 분석법을 기준으로 생각해보세요!

# [실습 풀이] Unit Test 근거 설명하기

## [문제 1 답안]

원칙	적용	이유
<b>Fast</b>	✓	단순 덧셈 연산만 수행하므로 <b>밀리초 이내</b> 실행 완료
<b>Isolated</b>	✓	외부 DB, 파일, 네트워크에 <b>의존하지 않음</b>
<b>Repeatable</b>	✓	<code>add(2, 3)</code> 은 <b>항상 5</b> 를 반환 (순수 함수)
<b>Self-validating</b>	✓	<code>assertEqual(result, 5)</code> 로 <b>자동 판정</b> 가능
<b>Timely</b>	✓	함수 작성 직후 <b>바로 테스트</b> 작성 가능

## [문제 2 답안] 이 함수를 테스트 할 때 어떤 입력값을 테스트 해야 할까요?



### 경계값 분석 (Boundary Value Analysis, BVA)을 적용합니다

- 양수 + 양수:  $\text{add}(2, 3) = 5$
- 음수 + 음수:  $\text{add}(-2, -3) = -5$
- 양수 + 음수:  $\text{add}(5, -3) = 2$
- 0 포함:  $\text{add}(0, 5) = 5$ ,  $\text{add}(0, 0) = 0$
- 큰 수:  $\text{add}(1000000, 1000000) = 2000000$

## [문제 3 답안] 테스트 케이스는 최소 몇 개가 필요할까요?



최소 5개를 생각해 볼 수 있다!

```
def test_add_positive_numbers(self):      # 양수 + 양수
def test_add_negative_numbers(self):       # 음수 + 음수
def test_add_mixed_numbers(self):          # 양수 + 음수
def test_add_with_zero(self):              # 0 포함
def test_add_large_numbers(self):          # 큰 수
```



팁: 테스트 케이스 이름만 봐도 무엇을 테스트하는지 알 수 있어야 합니다!



# Python `unittest`

## `unittest` 단어 설명



**unit**

라틴어 *unus* (하나)

→ **최소 단위**

프로그래밍에서 함수/메서드



**test**

라틴어 *testum* (도가니)

→ **검증 도구**

코드의 정확성 확인



**unittest** = Python 표준 라이브러리에 포함된 **단위 테스트 프레임워크**

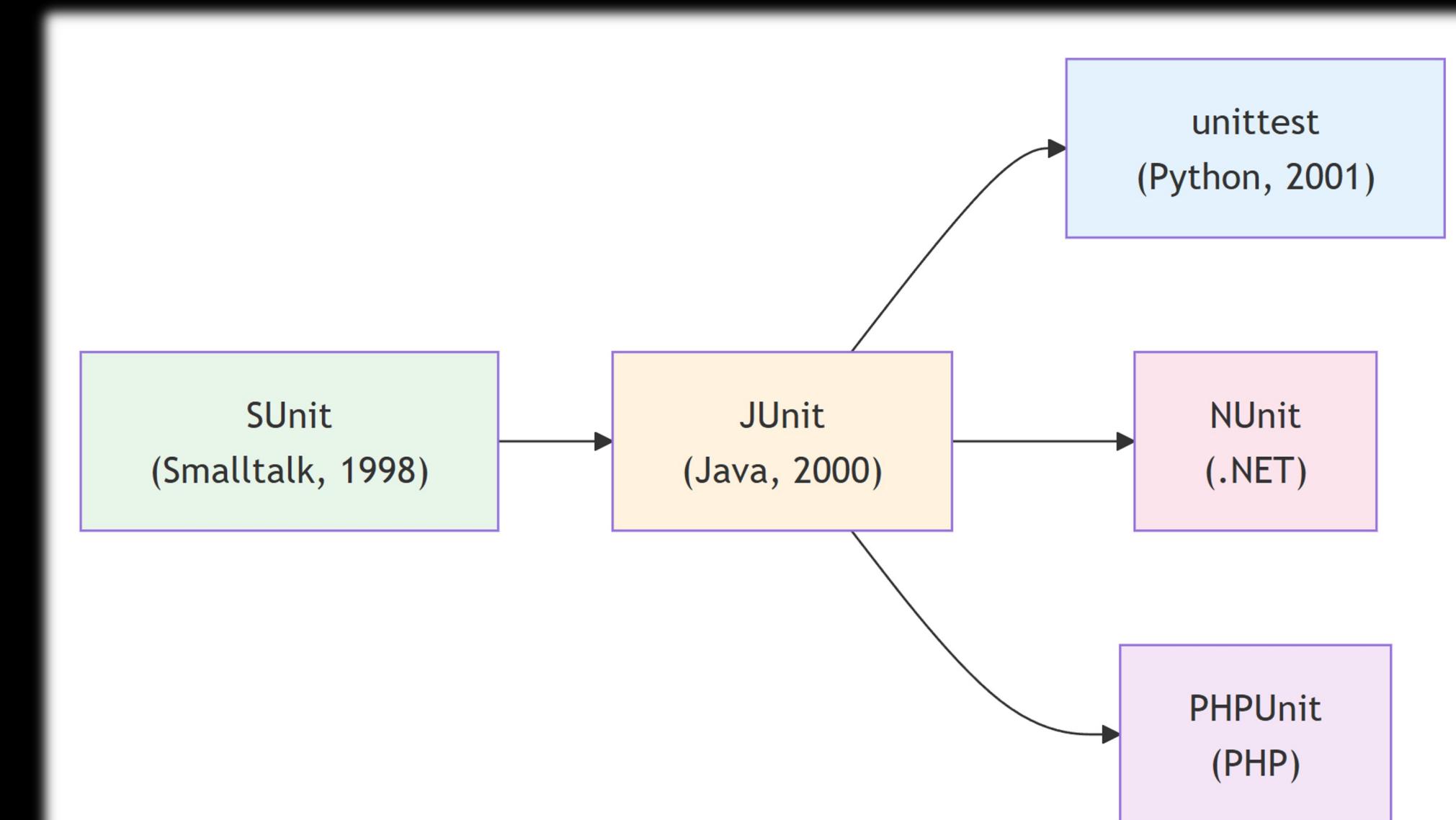
별도 설치 없이 `import unittest`로 바로 사용 가능!



## unittest의 역사

SUnit을 거쳐 JUnit에서 영감을 받은 unittest

- 서로 같은 계통이기 때문에 모든 xUnit Framework가 동일한 패턴을 공유한다.
  - TestCase, setup, teardown, assertion (오늘 뒤에서 배울 예정)
- 따라서 하나를 배우면 다른 언어의 테스트도 쉽게 진행 할 수 있다.





# Python `unittest`

## unittest의 역사

JUnit에서 영감을 받은 unittest

→ xUnit Family를 이해하면 언어가 바뀌어도 테스트 작성법은 동일하다는 걸 알게 된다.

연도	주요 사건
1998년	Kent Beck이 Smalltalk용 SUnit 개발
2000년	Kent Beck & Erich Gamma가 Java용 JUnit 개발
2001년	Steve Purcell이 Python용 unittest (PyUnit) 개발
2001년~	Python 2.1부터 표준 라이브러리로 포함



### "xUnit Family"

SUnit에서 시작된 테스트 프레임워크 설계 패턴을 따르는 프레임워크들을 통칭합니다.



# Python `unittest`

## **unittest = "시험지 양식"**

### 시험지 양식의 역할

- 수험번호, 이름 작성란 제공
- 문제 번호와 배치 통일
- 답안 작성 공간 규격화
- 채점 기준 명확화

### unittest의 역할

- TestCase 클래스 템플릿 제공
- 테스트 메서드 명명 규칙
- Assertion 메서드 표준화
- Pass/Fail 판정 기준 명확화



**핵심:** unittest는 "어떻게 테스트를 작성할 것인가"에 대한 **표준화된 방법**을 제공합니다.

모두가 같은 양식을 사용하면, 누구나 쉽게 이해할 수 있습니다.



## 4가지 핵심 구성요소

(1) TestCase → (2) TestSuite → (3) TestLoader → (4) TestRunner

### 1. TestCase (테스트 케이스)

기본 단위이며 개별 테스트 메서드들을 그룹화합니다.

`unittest.TestCase` 를 상속받아 사용합니다.

### 2. TestSuite (테스트 스위트)

여러 TestCase를 묶음으로 관리합니다.

관련된 테스트들을 함께 실행할 때 사용합니다.

### 3. TestLoader (테스트 로더)

TestCase를 자동으로 발견하고 로드합니다.

`test_`로 시작하는 메서드를 모두 찾아냅니다.

### 4. TestRunner (테스트 러너)

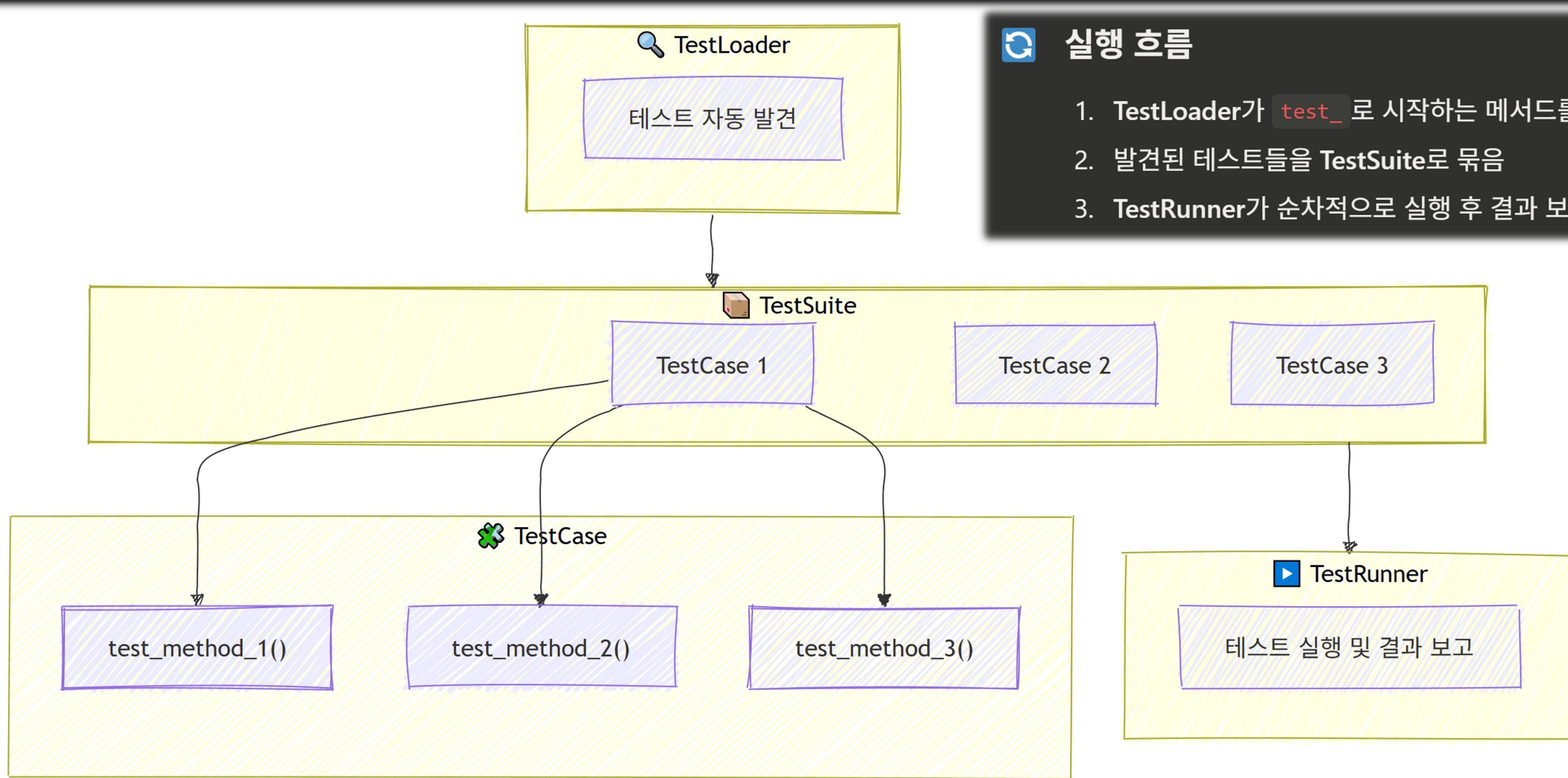
테스트를 실행하고 결과를 보고합니다.

콘솔에 Pass/Fail 결과를 출력합니다.



# unittest 핵심 구성요소

## unittest의 실행흐름



## 가장 기본적인 unittest 코드

Pycharm 혹은 VSCode에 직접 타이핑해서 만들어보세요!

→ 파일명은 "test\_sample.py" 혹은 "test\_{english\_name}.py"로 만들어보세요!

```
1 import unittest # 1. unittest 모듈 import
2
3
4 ▶ class TestMyFunction(unittest.TestCase): # 2. TestCase 상속
5
6 ▶     def test_example(self): # 3. test_로 시작하는 메서드
7         result = 1 + 1
8         self.assertEqual(result, 2) # 4. Assertion으로 검증
9
10
11 ▶ if __name__ == '__main__':
12     unittest.main() # 5. 테스트 실행
```

## 가장 기본적인 unittest 코드

앞서 본 코드를 자세하게 분석해보자

라인	코드	설명
1	import unittest	표준 라이브러리 불러오기
4	class ... (unittest.TestCase)	TestCase 클래스 상속
6	def test_example(self)	test_ 접두어 필수!
8	self.assertEqual(...)	기대값과 실제값 비교
12	unittest.main()	테스트 실행 진입점

# [간단 실습] unittest의 코드 구조

## 가장 기본적인 unittest 코드

이제 코드 작성이 완료 되었다면 실행해보자

test\_sample.py

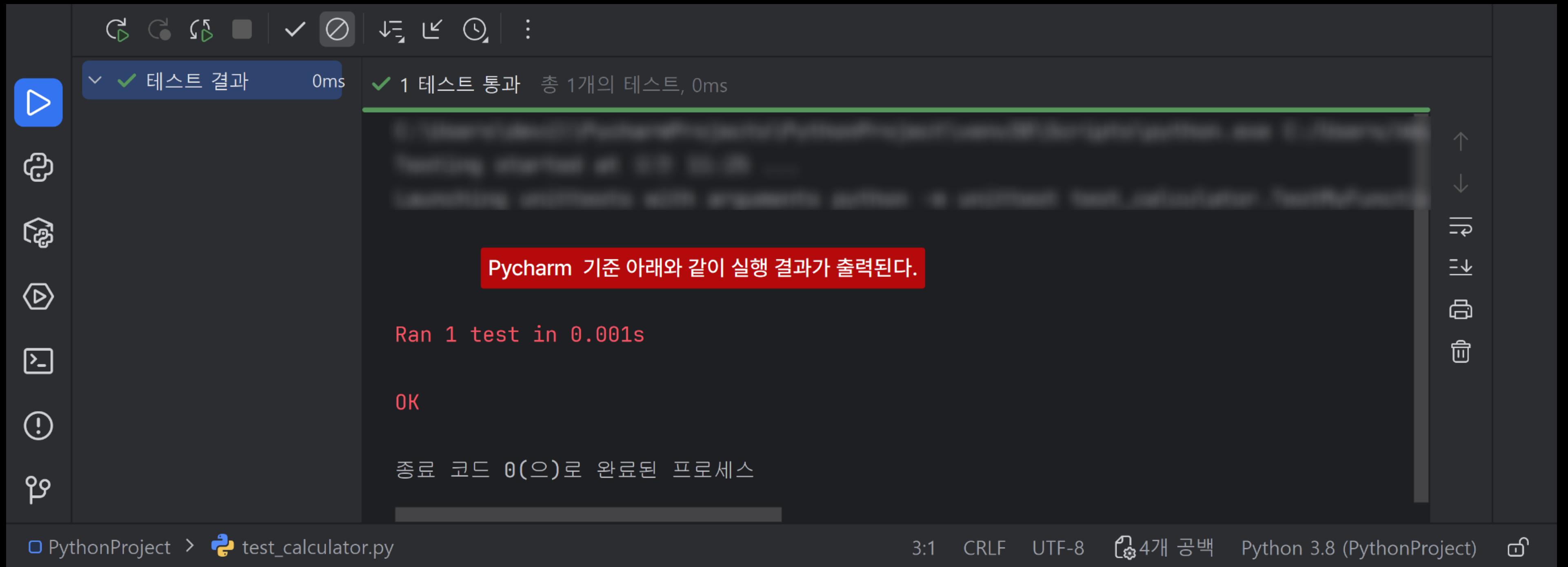
```
1 import unittest # 1. unittest 모듈 import
2
3
4 ⌂ ⌂ 'test_sample.TestMyFu...' 실행(U) Ctrl+Shift+F10 : # 2. TestCase 상속
5 ⌂ ⌂ 'test_sample.TestMyFu...' 디버그(D)
6 ⌂ ⌂ 커버리지로 'test_sample.TestMyFu...' 실행(V)
7 ⌂ ⌂ 프로파일 'test_sample.TestMyFu...' (E)
8 ⌂ ⌂ 'test_sample.TestMyFu...'에 대한 동시성 다이어그램(C)
9
10 실행 구성 수정...
11
12 on의 Python 테스트
```

PyCharm 기준으로  
작성한 테스트 코드 파일의 코드를 보면 "▶" 버튼이 있다.  
이 버튼을 눌러서 실행해보자

# [간단 실습] unittest의 코드 구조

## 가장 기본적인 unittest 코드

실행이 완료되었다면 아래와 같이 테스트가 정상 완료 된 것을 확인 할 수 있다



The screenshot shows the PyCharm interface with the 'test' tool window open. The status bar at the bottom indicates the project is named 'PythonProject' and the file is 'test\_calculator.py'. The test results window displays the following information:

- Toolbar icons: Refresh, Run, Stop, Checkmark, Circle with slash, Down arrow, Left arrow, Right arrow, Clock, Ellipsis.
- Section header: 테스트 결과 (Test Results) with 0ms.
- Summary: 1 테스트 통과 (1 test passed) in 0ms.
- Message: Pycharm 기준 아래와 같이 실행 결과가 출력된다. (The execution result is outputted as follows according to Pycharm's standard.)
- Output:

```
Ran 1 test in 0.001s
```

OK

종료 코드 0(으)로 완료된 프로세스
- Bottom status bar: 3:1 CRLF UTF-8 4개 공백 Python 3.8 (PythonProject) ⌂

## [실습] 테스트 클래스 작성하고 실행하기

아래 요구사항을 만족하는 `unittest` 코드를 작성하세요

1. `StringUtils` 라는 이름의 테스트 클래스를 만드세요
2. 슬라이드 아래에 적시된 세 개의 테스트 메서드를 작성하세요
3. 코드를 실행하고 결과를 확인해주세요

- `test_upper` : `"hello".upper()` 가 `"HELLO"` 인지 확인
- `test_lower` : `"WORLD".lower()` 가 `"world"` 인지 확인
- `test_strip` : `" hello ".strip()` 이 `"hello"` 인지 확인

4. 감이 잘 안오신다면 이 [템플릿 코드](#)를 사용해서 코드를 작성해주세요

## [실습] 테스트 클래스 작성하고 실행하기

실습 답안은 이 [링크](#)를 확인해주세요

- 링크가 눌리지 않는다면 아래 링크를 확인해주세요
- <https://kdt-gitlab.elice.io/-/snippets/16>
  
- 아래와 같이 실행 결과가 나오면 성공입니다.

```
...
-----
Ran 3 tests in 0.001s

OK
```

## [실습] 테스트 클래스 작성하고 실행하기

실습 결과를 풀이해서 해석하면 다음과 같습니다.

포인트	설명
클래스 이름	Test로 시작하는 이름 권장 ( <code>TestStringUtils</code> )
메서드 이름	반드시 <code>test_</code> 로 시작해야 자동 실행됨
Docstring	각 테스트의 목적을 명시하면 가독성 향상
<code>assertEqual</code>	두 값이 같은지 비교하는 가장 기본적인 Assertion



테스트 메서드 이름은 `test_` + `테스트하는_내용` 형식으로 작성하세요!

예를들면 아래와 같은 형식입니다

- `test_add_positive_numbers()`
- `test_login_with_valid_credentials()`



# TestCase 클래스 이해하기

## 'TestCase' 단어를 자세히 살펴봅시다.



Test (시험)

라틴어 *testum*

→ 검증하다

코드가 올바르게 동작하는지 확인



Case (경우, 상황)

라틴어 *casus* (떨어지다)

→ 발생할 수 있는 상황

특정 조건에서의 동작 시나리오



TestCase = "특정 상황에서 코드를 검증하는 단위"

예: "양수 덧셈 상황", "음수 덧셈 상황", "0을 포함한 상황"



# TestCase 클래스 이해하기



TestCase = "수능 시험 문제지 한 세트"

## 수능 시험 문제지

- 한 과목의 모든 문제를 포함
- 각 문제는 독립적으로 풀 수 있음
- 채점 기준이 명확함
- 전체 점수로 합산됨

## TestCase 클래스

- 관련 테스트 메서드들을 그룹화
- 각 테스트는 독립적으로 실행 가능
- Pass/Fail 기준이 명확함
- 전체 결과로 합산됨



예시를 들어보겠습니다.

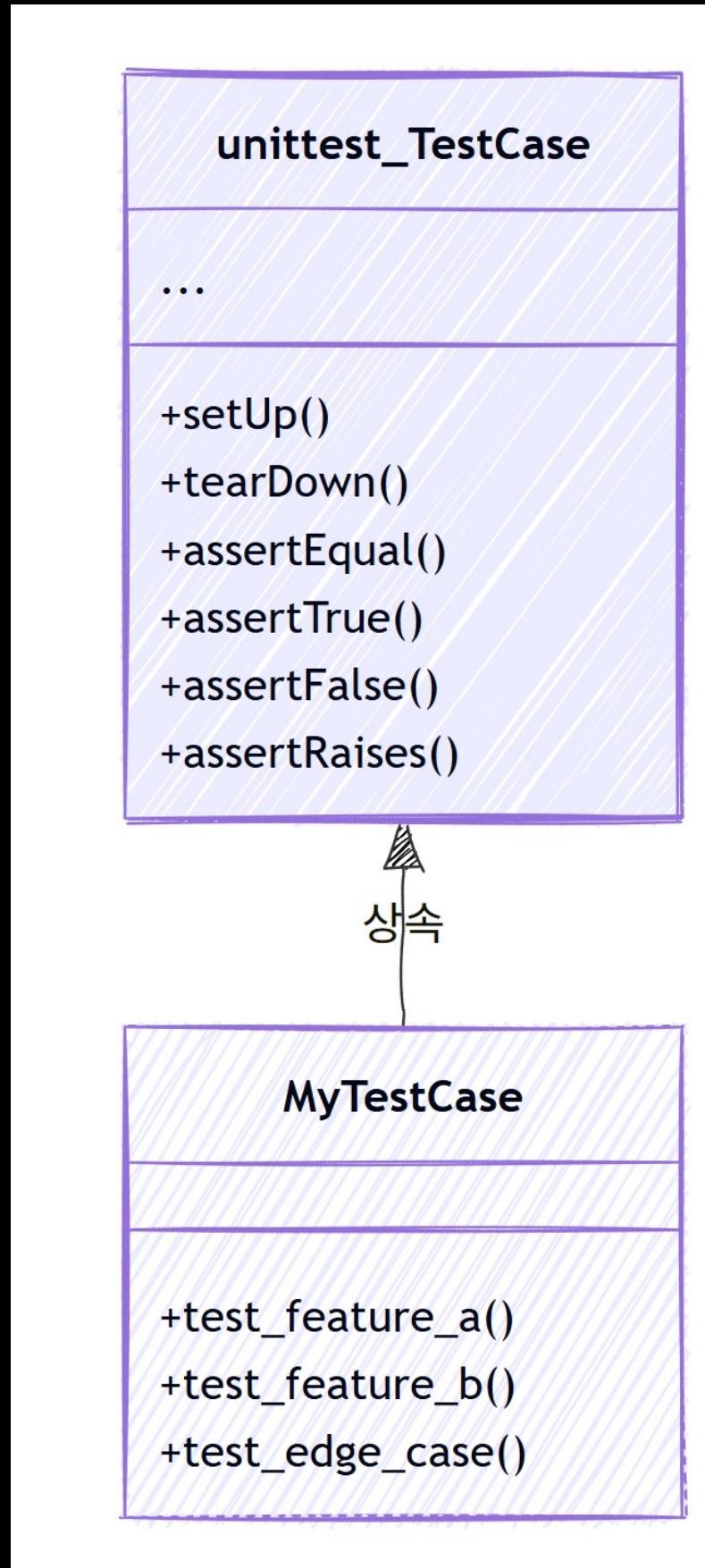


- TestCalculator = "계산기 과목 시험지"
  - test\_add = "1번 문제: 덧셈"
  - test\_subtract = "2번 문제: 뺄셈"
  - test\_multiply = "3번 문제: 곱셈"



# TestCase 클래스 이해하기

## 'unittest.TestCase' 상속 구조



### 💡 상속이란?

상속(Inheritance)은 기존 클래스의 기능을 물려받아 새로운 클래스를 만드는 것입니다.

**비유:** 부모님의 재산을 물려받는 것과 비슷합니다.

- **부모 클래스** (`unittest.TestCase`): 이미 만들어진 기능들 (`assertEqual`, `setUp` 등)
- **자식 클래스** (`MyTestCase`): 부모의 기능을 물려받고, 자신만의 테스트 메서드 추가

**장점:** 이미 만들어진 코드를 다시 작성할 필요 없이 바로 사용할 수 있습니다!

### 📚 상속을 통해 얻는 것들

1. **Assertion 메서드:** `assertEqual`, `assertTrue`, `assertRaises` 등
2. **생명주기 메서드:** `setUp`, `tearDown`
3. **테스트 실행 기능:** 자동 탐색 및 실행



# TestCase 클래스 이해하기

## "상속"의 핵심요약 및 한줄 정리

상속 → 남이 만든 기능을 내 것처럼 쓰는 것!

### 상속이 없다면?

모든 기능을 직접 만들어야 함  
✗ 시간이 오래 걸림  
✗ 실수하기 쉬움

### 상속을 사용하면?

이미 만들어진 기능을 가져다 쓰기  
✓ 빠르게 개발 가능  
✓ 검증된 기능 활용

### 코드 예시

```
class MyTest(unittest.TestCase): # ← unittest.TestCase의 기능을 물려받음
    def test_a(self):
        self.assertEqual(..., ...) # ← 내가 안 만들었는데 사용 가능!
```



# TestCase 클래스 이해하기

## 테스트 메서드 명명 규칙



**중요:** 테스트 메서드는 반드시 `test_`로 시작해야 합니다!

```
class TestCalculator(unittest.TestCase):

    def test_add(self):          # ✓ 실행됨
        pass

    def test_subtract(self):     # ✓ 실행됨
        pass

    def add_test(self):         # ✗ 실행 안 됨!
        pass

    def helper_method(self):    # ✗ 실행 안 됨 (헬퍼 함수로 사용 가능)
        pass
```



# TestCase 클래스 이해하기

## 좋은 테스트 메서드 이름 예시

반드시 "test\_"로 시작하되 상황을 자세히 설명한 이름을 사용해야 한다.  
테스트 메서드명에 한글을 허용하는 곳도 있다.  
→ "test\_양수를\_더한다()" 와 같이 한글을 쓰는 곳도 있다.

메서드 이름	설명
test_add_positive_numbers()	양수 덧셈 테스트
test_login_with_valid_credentials()	유효한 자격증명으로 로그인
test_divide_by_zero_raises_error()	0으로 나누면 에러 발생
test_1()	✗ 무엇을 테스트하는지 알 수 없음

## 계산기 클래스와 TestCase

계산기 클래스를 통해 TestCase를 다시 한번 살펴보자

우선 계산기 클래스를 다음과 같이 만들자

```
# calculator.py
class Calculator:
    def add(self, a, b):
        return a + b

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

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("나눠지는 0이 될 수 없습니다")
        return a / b
```

## 계산기 클래스와 TestCase

방금 만든 계산기 클래스의 작동을 보장하기 위한 테스트 코드를 만들자.

```
# test_calculator.py
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):

    def test_add(self):
        calc = Calculator()
        self.assertEqual(calc.add(2, 3), 5)

    def test_subtract(self):
        calc = Calculator()
        self.assertEqual(calc.subtract(5, 3), 2)

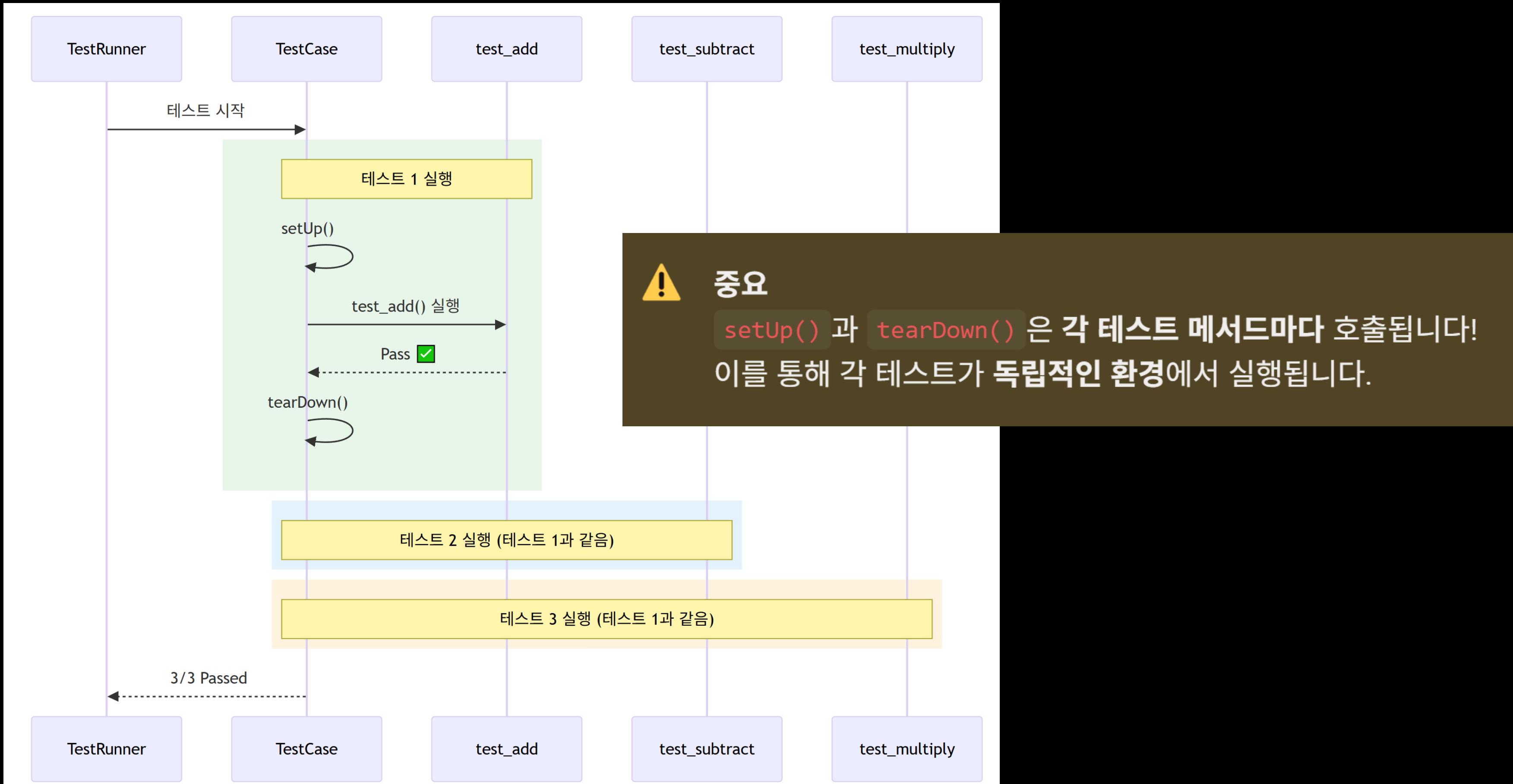
    def test_multiply(self):
        calc = Calculator()
        self.assertEqual(calc.multiply(4, 3), 12)

    def test_divide(self):
        calc = Calculator()
        self.assertEqual(calc.divide(10, 2), 5)

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

# [간단 실습] TestCase 클래스 이해하기

## TestCase 내 테스트 메서드 실행 순서





# TestCase 클래스 이해하기

## TestCase 그룹화 전략

# ✓ 좋은 예: 기능별로 분리

```
class TestCalculatorAdd(unittest.TestCase):
    """Calculator의 add 메서드 테스트"""
    def test_add_positive(self): ...
    def test_add_negative(self): ...
    def test_add_zero(self): ...
```

```
class TestCalculatorDivide(unittest.TestCase):
    """Calculator의 divide 메서드 테스트"""
    def test_divide_normal(self): ...
    def test_divide_by_zero(self): ...
```

# ✗ 나쁜 예: 모든 것을 한 클래스에

```
class TestEverything(unittest.TestCase):
    def test_add(self): ...
    def test_divide(self): ...
    def test_login(self): ... # 관련 없는 기능!
    def test_send_email(self): ... # 관련 없는 기능!
```



### 규칙

하나의 TestCase는 **하나의 클래스 또는 기능을 테스트합니다.**



# TestCase 클래스 이해하기

## 왜 테스트는 그룹화처럼 격리되어서 관리되어야 할까?

### ✗ 격리되지 않은 테스트

```
class TestBad(unittest.TestCase):
    counter = 0 # 클래스 변수 공유!

    def test_first(self):
        self.counter += 1
        self.assertEqual(self.counter, 1)

    def test_second(self):
        self.counter += 1
        # 실패! counter가 2가 됨
        self.assertEqual(self.counter, 1)
```

테스트 실행 순서에 따라 결과가 달라짐!

### ✓ 격리된 테스트

```
class TestGood(unittest.TestCase):

    def setUp(self):
        self.counter = 0 # 매번 초기화

    def test_first(self):
        self.counter += 1
        self.assertEqual(self.counter, 1)

    def test_second(self):
        self.counter += 1
        # 성공! counter가 1
        self.assertEqual(self.counter, 1)
```

setUp()으로 매번 새로운 상태로 시작!



## 왜 테스트는 그룹화처럼 격리되어서 관리되어야 할까?

격리의 원칙 세가지를 꼭 기억하자

1. 각 테스트는 다른 테스트에 영향을 주지 않아야 한다.
2. 테스트 실행 순서가 바뀌어도 동일한 결과가 나와야 한다.
3. 외부 상태 (DB, 파일)에 의존하지 않아야 한다.



## [실습] 문자열 처리 함수 3개를 테스트 하기

# 문자열 처리 함수 세 개를 테스트하는 TestCase 작성하기



## 문제

아래 `StringProcessor` 클래스의 메서드들을 테스트하는 TestCase를 작성하세요.

```
# string_processor.py
class StringProcessor:

    def reverse(self, text):
        """문자열을 반전합니다"""
        return text[::-1]

    def count_vowels(self, text):
        """모음(a, e, i, o, u) 개수를 세니다"""
        return sum(1 for char in text.lower() if char in 'aeiou')

    def is_palindrome(self, text):
        """회문인지 확인합니다 (공백/대소문자 무시)"""
        cleaned = text.replace(" ", "").lower()
        return cleaned == cleaned[::-1]
```

## 요구사항

1. `TestStringProcessor` 클래스를 만드세요
2. 각 메서드당 최소 2개 이상의 테스트 케이스를 작성하세요
3. 경계값 (빈 문자열 등)도 테스트하세요

## [실습] 문자열 처리 함수 3개를 테스트 하기

# 문자열 처리 함수 세 개를 테스트하는 TestCase 작성하기

## ? 문제

아래 `StringProcessor` 클래스의 메서드들을 테스트하는 TestCase를 작성하세요.

- 테스트 대상 코드는 [링크](#)에서 확인 할 수 있습니다. (직접 타이핑하시는 것을 권장드립니다)
- 클릭이 안된다면 아래 링크를 복사해주세요
- <https://kdt-gitlab.elice.io/-/snippets/17>

## [실습] 문자열 처리 함수 3개를 테스트 하기

### 문자열 처리 함수 세 개를 테스트하는 TestCase 작성하기

실습 답안은 이 [링크](#)를 확인해주세요.

→ 클릭이 안되는 경우 <https://kdt-gitlab.elice.io/-/snippets/18> 를 복사해주세요.

실행 결과는 아래와 같이 나오면 성공입니다.

→ 총 실행 시간 (0.002s) 은 하드웨어 상태에 따라 상이 합니다.

```
.....
```

```
-----  
Ran 9 tests in 0.002s
```

```
OK
```



## [실습] 문자열 처리 함수 3개를 테스트 하기

# 문자열 처리 함수 세 개를 테스트하는 TestCase 작성하기

풀이 해설

 테스트 메서드는 **하나의 동작만** 테스트합니다.  
하나의 테스트에서 여러 가지를 검증하면 실패 시 원인 파악이 어려워집니다.

포인트	설명
setUp 사용	각 테스트에서 공통으로 사용하는 인스턴스를 <code>setUp()</code> 에서 생성
경계값 테스트	빈 문자열, 한 글자 등 옛지 케이스 포함
메서드 그룹화	주석으로 각 메서드별 테스트를 구분
다양한 Assertion	<code>assertEqual</code> , <code>assertTrue</code> , <code>assertFalse</code> 활용



## 'Assert'의 어원을 알아보자

Assert는 라틴어 assere에서 유래했습니다.

- ad- (to, 향하여) + serere (join, 연결하다)
- 원래의미 : "자신의 주장에 연결하다" → 단언하다, 주장하다



프로그래밍에서의 의미: "이 값이 그렇다고 단언한다. 아니라면 테스트 실패!"

## 일상에서의 Assert

- 법정 → "피고인이 범인이라고 주장합니다" (증거 제시 필요)
- 수학 → "1 + 1 = 2라고 단언합니다"
- 테스트 → "add(1, 1)의 결과가 2라고 단언한다"



# Assertion 메서드에 대해서

## Assertion = 법정에서의 증거 제시

테스트에서 Assertion은 법정에서 증거를 제시하는 것과 같습니다.

법정	테스트	핵심 요약
검사	테스트 코드	좋은 검사(테스트)는 명확하고 구체적인 증거(Assertion)를 제시합니다. "뭔가 잘못됐다"가 아니라 "정확히 무엇이 어떻게 잘못됐는지" 보여줘야 합니다.
피고인	테스트 대상 코드	
증거 제시	Assertion 메서드 호출	
"피고인이 현장에 있었다!"	<code>assertEqual(result, expected)</code>	
<input checked="" type="checkbox"/> 증거 채택 → 유죄 판결	<input checked="" type="checkbox"/> 테스트 통과 (Pass)	
<input type="checkbox"/> 증거 기각 → 무죄 방면	<input type="checkbox"/> 테스트 실패 (Fail)	



# Assertion 메서드에 대해서

## 자주 사용하는 기본 Assertion 메서드

메서드	검증 내용	사용 예시
<code>assertEqual(a, b)</code>	<code>a == b</code> 인지 확인	값이 같은지 비교
<code>assertNotEqual(a, b)</code>	<code>a != b</code> 인지 확인	값이 다른지 비교
<code>assertTrue(x)</code>	<code>x</code> 가 <code>True</code> 인지 확인	조건이 참인지 검증
<code>assertFalse(x)</code>	<code>x</code> 가 <code>False</code> 인지 확인	조건이 거짓인지 검증
<code>assertIs(a, b)</code>	<code>a is b</code> (동일 객체)	같은 객체인지 확인
<code>assert IsNot(a, b)</code>	<code>a is not b</code>	다른 객체인지 확인
<code>assertIsNone(x)</code>	<code>x is None</code>	<code>None</code> 인지 확인
<code>assert IsNotNone(x)</code>	<code>x is not None</code>	<code>None</code> 이 아닌지 확인



### Tip

`assertEqual` 만으로도 대부분의 테스트가 가능해요.

하지만 더 구체적인 메서드를 사용하면 실패 메시지가 더 명확해집니다.



## [간단 실습] Assertion 메서드에 대해서

### 기본 Assertion 활용하기 (실습하고 결과보기)

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

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/19> 를 복사해주세요.

# [간단 실습] Assertion 메서드에 대해서

## 기본 Assertion 활용하기 (실습하고 결과보기)

- 코드를 실행하면 아래와 같은 결과를 얻을 수 있다.

The screenshot shows the PyCharm interface with the 'test\_sample.py' file selected in the 'Python Tests' tab. The 'Test Runner' window displays the execution results:

- 테스트 결과:** 5 테스트 통과 (총 5개의 테스트, 1ms)
- 테스트 목록:** test\_sample > TestBasicAssertions
  - test\_assertEqual\_with\_numbers (숫자 비교: assertEquals) - 1ms
  - test\_assertEqual\_with\_strings (문자열 비교: assertEquals) - 0ms
  - test\_assertIsNone (None 검증: assertIsNone) - 0ms
  - test\_assertIs\_same\_object (동일 객체 검증: assertIs) - 0ms
  - test\_assertTrue\_and\_assertFalse (불리언 검증: assertTrue, assertFalse) - 0ms

At the bottom, the output shows: `Ran 5 tests in 0.001s` and `OK`. A status message at the bottom right says: `종료 코드 0(으)로 완료된 프로세스`.



## 다양한 자료구조 (컬렉션) 와 예외를 위한 Assertion

메서드	검증 내용
<code>assertIn(a, b)</code>	a가 b 안에 포함되어 있는지 (a in b)
<code>assertNotIn(a, b)</code>	a가 b 안에 없는지 (a not in b)
<code>assertInstanceOf(a, b)</code>	a가 b 타입의 인스턴스인지
<code>assertGreater(a, b)</code>	$a > b$ 인지 확인
<code>assertLess(a, b)</code>	$a < b$ 인지 확인
<code>assertGreaterEqual(a, b)</code>	$a \geq b$ 인지 확인
<code>assertAlmostEqual(a, b)</code>	$a \approx b$ (부동소수점 비교)
<code>assertRaises(Exception)</code>	특정 예외가 발생하는지 확인
<code>assertRaisesRegex(Exception, regex)</code>	예외 + 메시지 패턴 확인



## 다양한 자료구조 (컬렉션) 와 예외를 위한 Assertion



### 부동소수점 비교 주의

`0.1 + 0.2 == 0.3` 은 `False` 입니다!

이는 컴퓨터가 부동소수점을 이진수로 저장하면서 발생하는 **미세한 오차** 때문입니다.

부동소수점은 반드시 `assertAlmostEqual` 을 사용하세요.

예: `0.1 + 0.2` 는 실제로 `0.3000000000000004` 로 저장됩니다.



## assertRaises로 예외 테스트

### 예외가 발생하는지 테스트하기

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

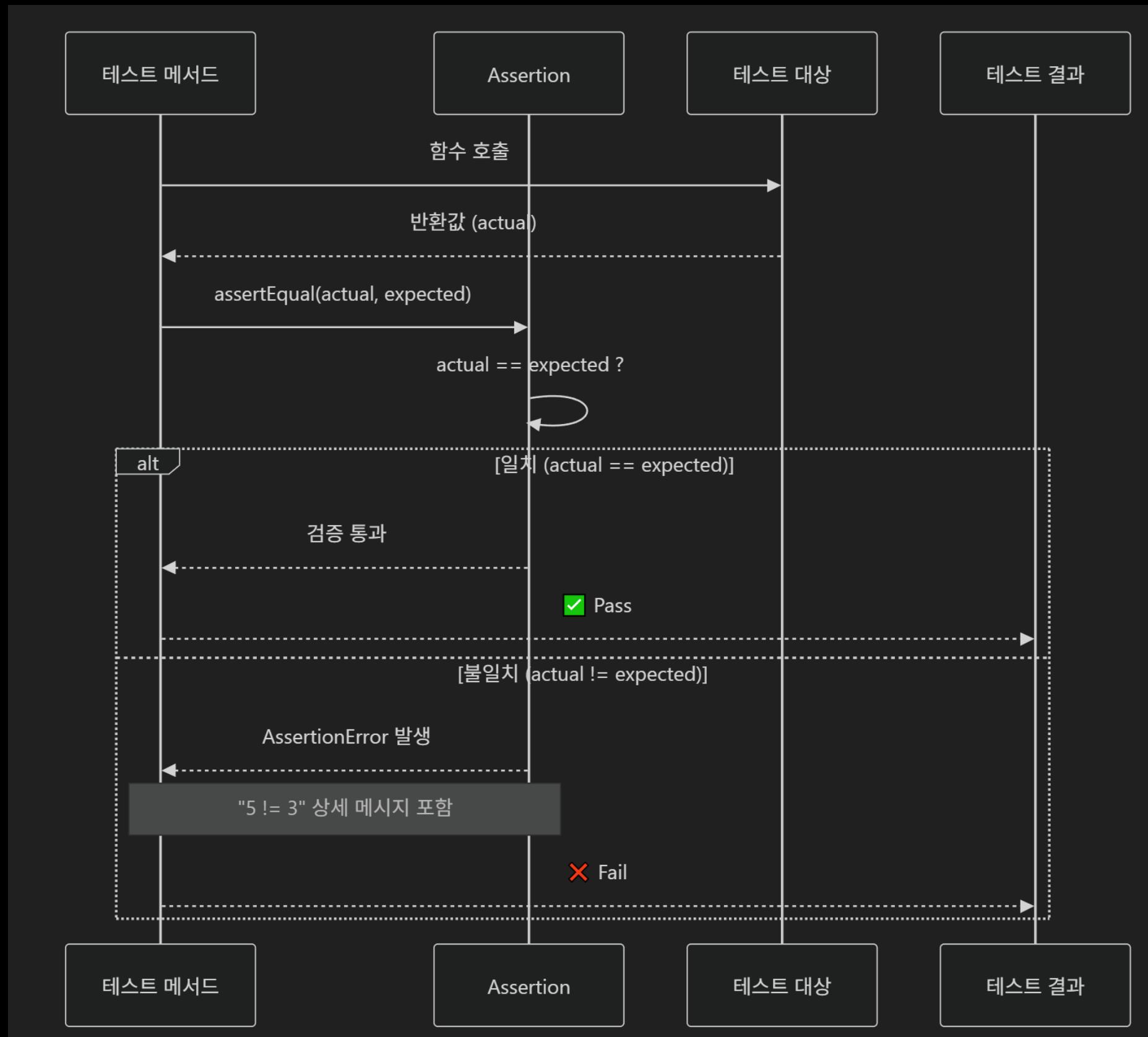
→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/20> 를 복사해주세요.



# Assertion 메서드에 대해서

## Assertion 성공/실패 분기 흐름을 살펴보자



### 흐름 요약

1. 테스트 메서드가 테스트 대상 코드를 호출
2. 반환값(actual)을 받음
3. Assertion 메서드로 기대값(expected)과 비교
4. 일치하면 Pass, 불일치하면 Fail + 상세 메시지



# setUp / tearDown 생명주기

## setUp과 teardown의 어원을 살펴보자

Set + Up

- Set: 놓다, 배치하다
- Up: 위로, 완성 상태로
- 결합하면? → 무언가를 제대로 배치해서 준비 완료 상태로 만든다!

Tear + Down

- Tear: 찢다, 분리하다
- Down: 아래로, 해체 방향으로
- 결합하면? → 세워진 것을 분해하여 원래 상태로 되돌린다



### 실제 세계에서의 예시

- 이벤트 **set up**: 무대, 음향, 조명 설치
- 이벤트 **tear down**: 행사 후 모든 장비 철거



# setUp / tearDown 생명주기

## 요리하는 과정을 통해 테스트의 생명주기를 살펴보자

요리 과정	테스트 과정	해당 메서드
🥕 재료 준비	테스트 데이터/객체 생성	setUp()
🔍 요리하기	실제 테스트 실행	test_XXX()
🧹 설거지	리소스 정리/초기화	tearDown()



### 핵심 요약

좋은 요리사는 매 요리 전에 **깨끗한 도마와 신선한 재료**를 준비하지요!

요리 후에는 **다음 요리를 위해 정리합니다!** 테스트도 당연히 똑같답니다!



# setUp / tearDown 생명주기

## 왜 매번 새로 준비해야 하는 걸까?

테스트를 "격리"하지 않으면 테스트 상호간에 영향이 갈 수 밖에 없어서 일관된 테스트가 불가능하다!

```
# 잘못된 예: 한 테스트가 다른 테스트에 영향
class BadTest(unittest.TestCase):
    items = [] # 클래스 변수 - 모든 테스트가 공유!

    def test_add_item(self):
        self.items.append("apple")
        self.assertEqual(len(self.items), 1) # 처음엔 통과

    def test_add_another(self):
        self.items.append("banana")
        self.assertEqual(len(self.items), 1) # 실패! 이미 apple이 있음
```



# setUp / tearDown 생명주기

## 왜 매번 새로 준비해야 하는 걸까?

### 테스트 격리(Test Isolation)의 중요성

`setUp` 으로 매 테스트마다 새로운 리스트를 만들면 해결됩니다!

### 왜 격리가 필요할까요?

- 각 테스트는 **독립적**이어야 합니다 - 다른 테스트의 결과에 영향을 받으면 안 됩니다
- 테스트 실행 순서가 바뀌어도 **같은 결과**가 나와야 합니다
- 한 테스트가 실패해도 다른 테스트는 **정상적으로 실행**되어야 합니다

`setUp` 은 매 테스트마다 **깨끗한 상태**를 보장하는 핵심 메커니즘입니다!



# setUp / tearDown 생명주기

## 각 테스트 메서드마다 반복되는 생명주기



### 핵심 요약!

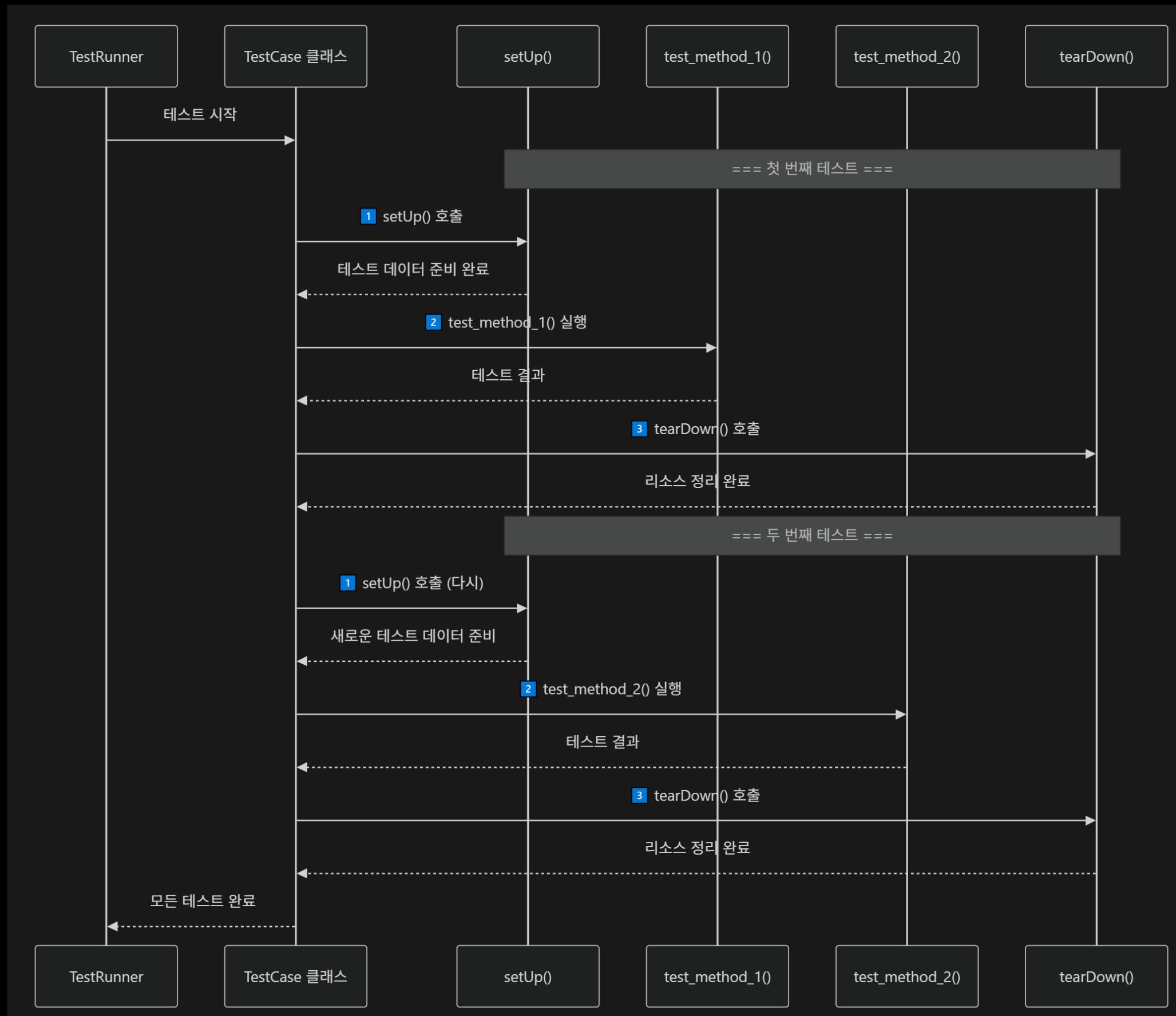
setUp 과 tearDown 은 각 테스트 메서드마다 실행됩니다.  
3개의 테스트가 있으면 각각 3번씩 호출됩니다!





# setUp / tearDown 생명주기

## 테스트 메서드 실행 순서를 자세히 살펴보자



다이어그램이 잘 보이지 않는다면?

- 이 [링크](#)에서 확인해주세요

링크가 보이지 않는다면 아래 링크를 복사해주세요.

- <https://kdt-gitlab.elice.io/-/snippets/21>



## [간단 실습] setUp / tearDown 생명주기

**매 테스트마다 새로운 객체가 생성되는 것을 코드로 확인하자 (setUp 활용)**

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

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/22> 를 복사해주세요.



# [간단 실습] setUp / tearDown 생명주기

매 테스트마다 새로운 객체가 생성되는 것을 코드로 확인하자 (setUp 활용)

실행 결과

실행 test\_sample.TestShoppingCart의 Python 테스트

테스트 결과

- 4 테스트 통과 총 4개의 테스트, 1ms
  - setUp: 새 장바구니 생성 테스트: 상품 추가
  - setUp: 새 장바구니 생성 테스트: 장바구니 비우기
  - setUp: 새 장바구니 생성 테스트: 총액 계산
  - setUp: 새 장바구니 생성 테스트: 초기 상품 개수

Ran 4 tests in 0.002s

OK

종료 코드 0(으)로 완료된 프로세스

`setUp`이 총 네번 호출 된 것을 확인하세요!



## [간단 실습] setUp / tearDown 생명주기

**매 테스트 후 리소스가 정리 되는 것을 확인하자(tearDown 활용)**

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

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/23> 복사해주세요.



# [간단 실습] setUp / tearDown 생명주기

## 매 테스트 후 리소스가 정리 되는 것을 확인하자(tearDown 활용)

### 실행 결과

The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says '실행 test\_sample.py 내 Python 테스트'. The tool window has several tabs at the top: '실행' (selected), '터미널', '디버그', '파일', '설정', '도움말'. Below the tabs are icons for running, stopping, and refreshing. The main area is divided into sections: '테스트 결과' (Test Results) and '터미널' (Terminal).  
**테스트 결과 (Test Results):**

- 2 테스트 통과 총 2개의 테스트, 10ms
  - setUp: 임시 파일 생성 - tmp...bzo4tb.txt
  - tearDown: 임시 파일 삭제 - tmp...bzo4tb.txt
  - setUp: 임시 파일 생성 - tmp...u4socz3\_.txt
- test\_sample
- TestFileProcessor
  - test\_append (파일 추가 쓰기 테스트)
  - test\_write\_and\_read (파일 쓰기/읽기 테스트)

**터미널 (Terminal):**

```
Ran 2 tests in 0.011s  
OK  
tearDown: 임시 파일 삭제 - tmp...u4socz3_.txt  
종료 코드 0(으)로 완료된 프로세스
```



# [간단 실습] setUp / tearDown 생명주기

## 매 테스트 후 리소스가 정리 되는 것을 확인하자(tearDown 활용)

### 실행 결과

The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says '실행 test\_sample.py 내 Python 테스트'. The tool window has tabs for '실행' and '터미널'. A large yellow warning box is centered in the window with the following text:

**⚠ 중요 사항!**  
리소스 누수를 방지하기 위해서  
`tearDown` 은 테스트가 실패하거나 에러가 발생해도 항상 실행됩니다!

Below the warning, the terminal output shows:

```
Ran 2 tests in 0.011s  
  
OK  
trash tearDown: 임시 파일 삭제 - tmpu4socz3_.txt  
  
종료 코드 0(으)로 완료된 프로세스
```



# setUp / tearDown 생명주기

## setUpClass / tearDownClass (클래스 레벨)

메서드 레벨 vs 클래스 레벨

구분	메서드 레벨	클래스 레벨
메서드명	setUp() / tearDown()	setUpClass() / tearDownClass()
실행 횟수	각 테스트마다	클래스당 1번
데코레이터	없음	@classmethod 필수
사용 시점	가벼운 객체 생성	무거운 리소스 (DB 연결 등)



## [간단 실습] setUp / tearDown 생명주기

### setUpClass / tearDownClass (클래스 레벨)

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

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/24> 복사해주세요.



## [간단 실습] setUp / tearDown 생명주기

### setUpClass / tearDownClass (클래스 레벨)

실행 후 결과 (직접 실행해 보고 해당 결과를 눈으로 보며 순서를 반드시 파악할 것)

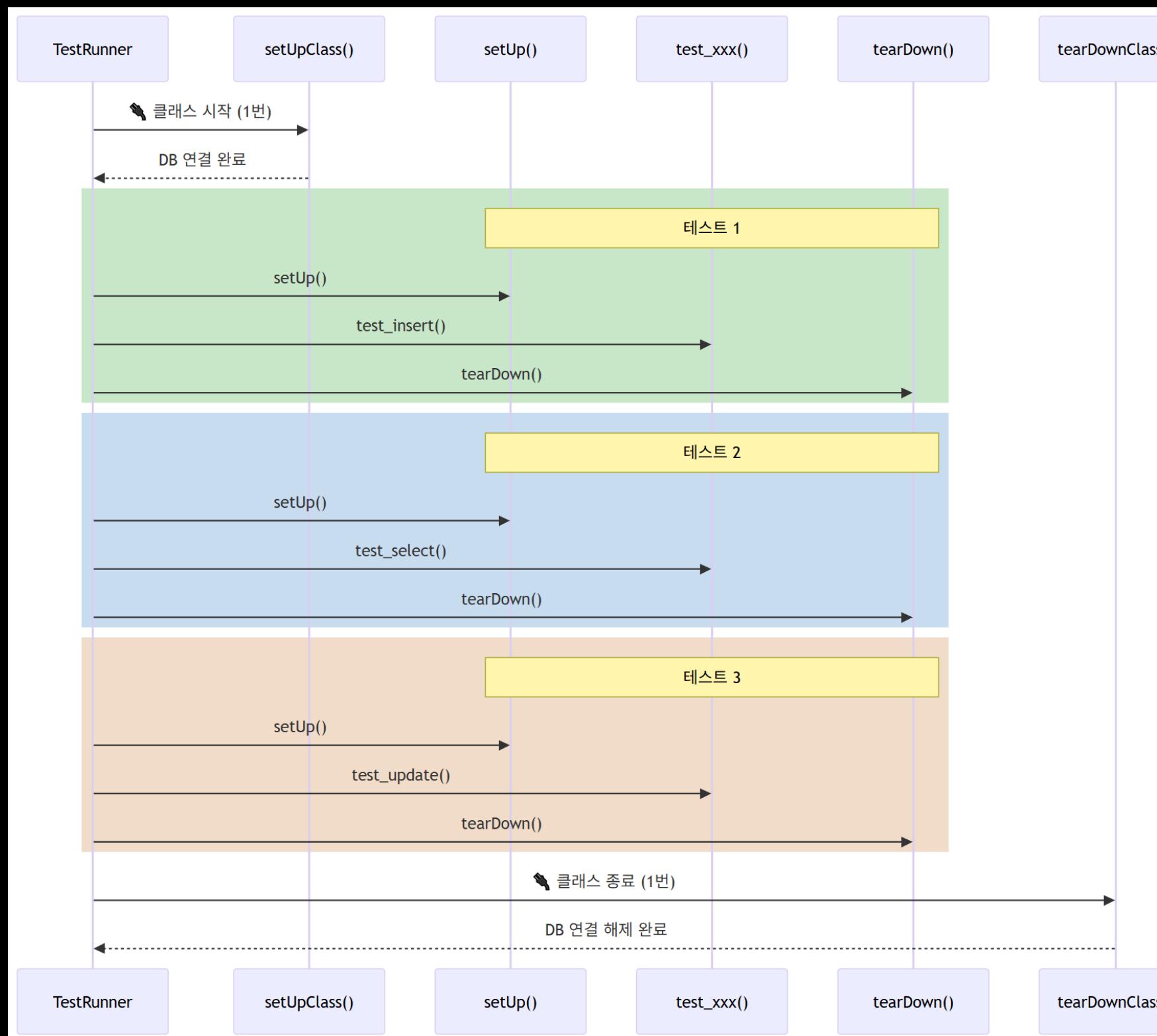
```
🔌 setUpClass: 데이터베이스 연결      ← 1번만!
📝 setUp: 트랜잭션 시작
    테스트: INSERT
📝 tearDown: 트랜잭션 롤백
📝 setUp: 트랜잭션 시작
    테스트: SELECT
📝 tearDown: 트랜잭션 롤백
📝 setUp: 트랜잭션 시작
    테스트: UPDATE
📝 tearDown: 트랜잭션 롤백
🔌 tearDownClass: 데이터베이스 연결 해제      ← 1번만!
```



# [실습 분석] setUp / tearDown 생명주기

## setUpClass / tearDownClass (클래스 레벨)

전체 생명주기를 시간 순서대로 살펴보자



다이어그램이 잘 보이지 않는다면?

- 이 [링크](#)에서 확인해주세요

링크가 보이지 않는다면 아래 링크를 복사해주세요.

- <https://kdt-gitlab.elice.io/-/snippets/25>



## [실습 분석] setUp / tearDown 생명주기

# setUpClass / tearDownClass (클래스 레벨)

## 사용 가이드라인



**메서드 레벨 ( `setUp` / `tearDown` ):** 기본값. 테스트 격리가 중요할 때

- 객체 인스턴스 생성
- 테스트 데이터 준비
- 상태 초기화



**클래스 레벨 ( `setUpClass` / `tearDownClass` ):** 리소스가 무거울 때

- 데이터베이스 연결/해제
- 서버 시작/종료
- 대용량 파일 로드



## 테스트를 실행하는 세 가지 방법

unittest는 여러 방법으로 실행할 수 있다.

방법	설명	사용 시점
커맨드라인	<code>python -m unittest</code> 명령어 사용	CI/CD, 스크립트 자동화
IDE 통합	PyCharm, VSCode 등에서 버튼 클릭	개발 중 빠른 피드백
스크립트 내장	<code>unittest.main()</code> 호출	단일 파일 테스트



## 테스트를 실행하는 세 가지 방법

### 방법 1: 커マン드라인 실행

```
# 모든 테스트 실행  
python -m unittest discover  
  
# 특정 모듈 실행  
python -m unittest test_calculator  
  
# 특정 클래스 실행  
python -m unittest test_calculator.TestAddition  
  
# 특정 메서드 실행  
python -m unittest test_calculator.TestAddition.test_positive_numbers
```



# unittest 실행과 결과 해석

## 테스트를 실행하는 세 가지 방법

### 방법 2: 스크립트 내장 실행

```
# test_calculator.py
import unittest

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(1 + 1, 2)

# 이 부분이 스크립트 내장 실행의 핵심!
if __name__ == '__main__':
    unittest.main()

# 직접 실행
python test_calculator.py
```



## 테스트 실행 결과 분석

### 방법 3: IDE 및 VSCode 내장 실행

The screenshot shows the VSCode interface with the Test Explorer and a code editor side-by-side.

**Test Explorer:** On the left, it displays a tree view of test results. A red callout box highlights the "Run Test" button (number 2) and the "Play" icon (number 1).

**Code Editor:** On the right, the code for `test_sample.py` is shown:

```
test_sample.py > TestResultTypes > test_pass_example
1 # test_results.py
2 import unittest
3
4
5 class TestResultTypes(unittest.TestCase):
6     """다양한 테스트 결과 유형을 보여주는 예제"""
7
8     def test_pass_example(self):
9         """성공하는 테스트 - 1 + 1은 항상 2"""
10        result = 1 + 1
11        self.assertEqual(result, 2)
12
13    def test_fail_example(self):
14        """실패하는 테스트 - 의도적으로 틀린 기대값 사용"""
15        result = 10 * 2
16        # 실제 결과는 20이지만, 기대값을 25로 설정
17        self.assertEqual(result, 25, "10 * 2는 25가 아닙니다!")
18
19    def test_error_example(self):
20        """에러 발생 테스트 - 0으로 나누기"""
21        numbers = [1, 2, 3]
22        # 존재하지 않는 인덱스 접근으로 에러 발생
23        value = numbers[10] # IndexError!
24
25    @unittest.skip("이 기능은 아직 구현되지 않았습니다")
26    def test_skip_example(self):
27        """건너뛰는 테스트 - 미구현 기능"""
28        # 아직 구현되지 않은 기능
29        self.assertTrue(False)
30
```



## 테스트 실행 결과 분석

### 결과 출력 예시

- 터미널에서 실행하는 경우 결과 문자열을 볼 수 있다. (읽는 방법은 다음 슬라이드)

The screenshot shows a terminal window with the following output:

```
$ python -m unittest test_sample.py
EF.s
=====
ERROR: test_error_example (test_sample.TestResultTypes)
에러 발생 테스트 - 0으로 나누기
-----
Traceback (most recent call last): (@ AI로 설명)
  File 'test_sample.py', line 23, in test_error_example
    value = numbers[10] # IndexError!
IndexError: list index out of range
=====
FAIL: test_fail_example (test_sample.TestResultTypes)
실패하는 테스트 - 의도적으로 틀린 기대 값 사용
```

A red arrow points from the 'E' in the output to a callout box containing the text: "테스트 결과 문자열 매 실행 시 순서는 다를 수 있다".

At the bottom left, it says "PythonProject > test\_sample.py". At the bottom right, it shows "21:20 CRLF UTF-8 4기".



# unittest 실행과 결과 해석

## 테스트 실행 결과 읽기

테스트 실행 후 출력되는 기호들은 각각의 의미가 있다



Fail과 Error는 다릅니다!

- Fail: 코드에 문제가 있음
- Error: 테스트 코드에 문제가 있음

기호	상태	의미
.	Pass (성공)	테스트가 예상대로 통과함
F	Fail (실패)	Assertion이 실패함 (기대값 ≠ 실제값)
E	Error (에러)	테스트 코드 자체에서 예외 발생
s	Skip (건너뜀)	@unittest.skip 데코레이터로 건너뜀
x	Expected Failure	예상된 실패 (정상)



## Verbose 모드란?

Verbose (장황한, 상세한)의 어원은 라틴어 verbosus로, "말이 많은" 이라는 뜻  
기본 모드와 다르게 각 테스트의 이름과 결과를 상세히 보여줌

### Verbose 모드 활성화

```
# -v 옵션 추가  
python -m unittest -v test_calculator
```

```
# 또는 스크립트에서  
if __name__ == '__main__':  
    unittest.main(verbosity=2)
```



## Verbose 모드 출력 비교

```
기본 모드
```

```
....  
-----  
Ran 4 tests in 0.001s  
OK
```

```
Verbose 모드 ( -v )
```

```
test_add (test_calculator.TestCalculator) ... ok  
test_subtract (test_calculator.TestCalculator) ... ok  
test_multiply (test_calculator.TestCalculator) ... ok  
test_divide (test_calculator.TestCalculator) ... ok  
-----  
Ran 4 tests in 0.001s  
OK
```



Tip: 개발 중에는 `-v` 옵션을 사용하고, CI/CD에서는 기본 모드를 사용하는 것이 일반적입니다.



## unittest 결과 해석 핵심 정리



1. **실행 방법:** 커맨드라인, IDE, 스크립트 내장 중 상황에 맞게 선택
2. **결과 기호:** . =성공, F =실패, E =에러, S =건너뛰됨
3. **Fail vs Error:** Fail은 코드 문제, Error는 테스트 문제
4. **Verbose 모드:** -v 옵션으로 상세 정보 확인



### 핵심 학습 포인트

1. Fail은 테스트 대상 코드의 문제 → 비즈니스 로직 수정 필요
2. Error는 테스트 코드의 문제 → 테스트 코드 수정 필요
3. Skip은 의도적인 건너뛰기 → 나중에 구현 예정
4. Verbose 모드는 어떤 테스트가 어떤 결과인지 한눈에 파악 가능



## Mock 의 단어를 풀어서 살펴보자

원래 의미 → 흉내내다, 모방하다

현대 의미 → 가짜의, 모조품의

 프로그래밍에서의 의미

실제 객체를 흉내 내는 가짜 객체를 만들어 테스트에 사용한다.

### 일상에서의 Mock

- 영화 촬영: 스탠트 더블 → 배우 대신 위험한 장면 연기
- 건축: 모형 건물 → 실제 건물 대신 축소 모형으로 검토
- 비행: 비행 시뮬레이터 → 실제 비행기 대신 모의환경



# unittest.mock에 대해서

## 왜 Mock이 필요할까?

Mock 없이 테스트하기 어려운 상황들

```
# 문제 상황 1: 외부 API 호출
def get_weather(city):
    response = requests.get(f"https://api.weather.com/{city}")
    return response.json()["temperature"]

# 문제 상황 2: 데이터베이스 연결
def get_user_balance(user_id):
    conn = database.connect()
    result = conn.execute(f"SELECT balance FROM users WHERE id={user_id}")
    return result.fetchone()[0]

# 문제 상황 3: 현재 시간 의존
def is_business_hours():
    current_hour = datetime.now().hour
    return 9 <= current_hour <= 18
```



# unittest.mock에 대해서

## 실제 객체로 테스트 할 때의 문제점

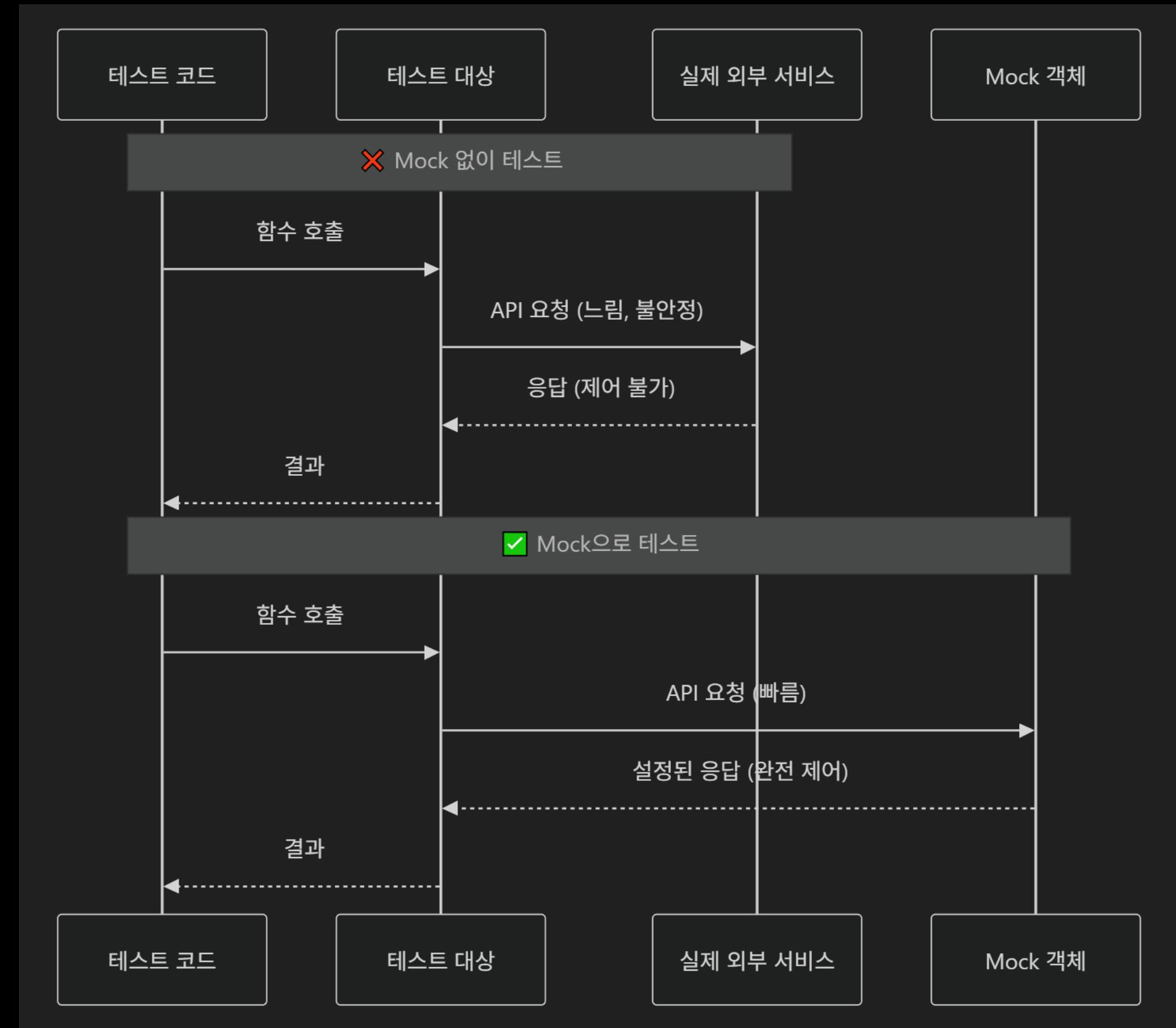
Mock 없이 테스트하기 어려운 상황들

문제	설명	Mock으로 해결
느림	네트워크, DB 연결에 시간 소요	즉시 응답하는 가짜 객체
불안정	외부 서비스 장애 시 테스트 실패	항상 일관된 응답 보장
비용	API 호출 비용, DB 리소스	무료로 무제한 테스트
제어 불가	에러 상황 재현 어려움	원하는 응답/에러 설정 가능
부작용	테스트가 실제 데이터 변경	실제 시스템에 영향 없음



# unittest.mock에 대해서

## Mock 없는 테스트 vs Mock 있는 테스트





## unittest.mock 핵심 구성요소

주요 클래스와 함수

구성요소	설명	사용 시점
Mock	기본 Mock 객체 - 모든 속성/메서드 자동 생성	간단한 가짜 객체 필요 시
MagicMock	Mock + 매직 메서드 지원 ( <code>len</code> , <code>iter</code> 등)	컨테이너, 컨텍스트 매니저 흉내
patch	기존 객체를 Mock으로 임시 교체	모듈/클래스의 특정 부분 교체
return_value	Mock 호출 시 반환할 값 설정	함수 반환값 지정
side_effect	호출 시 실행할 동작 (예외, 함수 등)	예외 발생, 동적 반환값



# unittest.mock 에 대해서

## Mock 기본 사용법

Mock 객체 생성과 설정

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

→ 링크가 안눌리는 경우 아래 링크를 복사해주세요

→ <https://kdt-gitlab.elice.io/-/snippets/26> 복사해주세요.



## Mock이 어떻게 호출되었는지 확인하기

```
from unittest.mock import Mock, call

mock = Mock()

# 여러 번 호출
mock.method(1, 2)
mock.method(3, 4)
mock.method("a", "b")

# 호출 검증 메서드들
mock.method.assert_called()          # 최소 1번 호출됨
mock.method.assert_called_once()      # 정확히 1번 호출됨 (실패!)
mock.method.assert_called_with("a", "b")  # 마지막 호출 인자 확인
mock.method.assert_any_call(1, 2)       # 어느 호출이든 이 인자로 호출됨

# 호출 횟수 확인
print(mock.method.call_count) # 3

# 모든 호출 내역 확인
print(mock.method.call_args_list) # [call(1, 2), call(3, 4), call('a', 'b')]

# 특정 순서로 호출되었는지 확인
mock.method.assert_has_calls([
    call(1, 2),
    call(3, 4),
    call("a", "b")
])
```

메서드	설명
assert_called()	최소 1번 이상 호출되었는지 확인
assert_called_once()	정확히 1번만 호출되었는지 확인
assert_called_with(*args, **kwargs)	마지막 호출이 특정 인자로 되었는지
assert_called_once_with(*args, **kwargs)	1번만 호출 + 특정 인자 확인
assert_any_call(*args, **kwargs)	어느 호출이든 특정 인자로 호출됨
assert_has_calls(calls, any_order=False)	특정 호출들이 순서대로 있는지
assert_not_called()	한 번도 호출되지 않았는지