

pytest를 활용한 코드레벨 테스트 가이드

(pytest: python 화이트박스 테스트 도구)

이 교육을 통해서,

- Python의 화이트박스 테스트 도구인 pytest의 기본 사용법을 배우고, 내 코드/프로젝트에 적용할 수 있습니다
- 작성한 테스트 코드를 이용하여 개발 IDE내에서 손쉽게 디버깅을 할 수 있습니다
- 테스트 커버리지, 단위테스트/통합 테스트의 개념과 이점을 이해합니다
- Mock 테스트가 필요한 경우를 이해하고, 내 개발 코드에 적용할 수 있습니다
- HTTP API에 대해서도 손쉽게 테스트 코드를 작성하고 확인할 수 있습니다

1. 개요

1.1 화이트박스 테스트란

1.2 pytest 개요

2. Pytest 기본 사용법

3. Mocking을 통한 단위 테스트

4. 코드레벨 통합 테스트(HTTP API 테스트)

5. 정리

코드 레벨, 화이트 박스 테스트란?



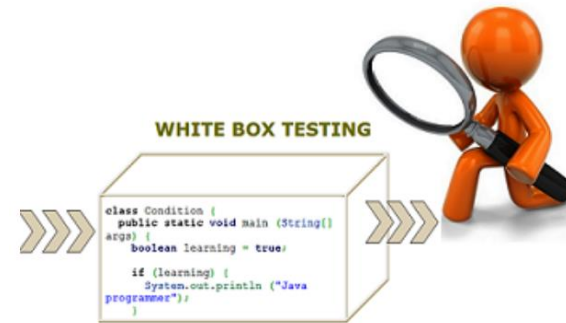
“코드레벨 테스트”?

- ※ 공식 용어는 아님
- 개발 코드 작성과 함께 바로, 빠르게 테스트 코드를 작성하고 기능 동작을 확인
- 코드 동작에 맞는 테스트 수행(=화이트박스 테스트)
- 개발 IDE 내에서 쉽고 빠르게 수행
- 테스트 코드를 짜면서 코드 리팩토링이 되기도 하고, 작성한 테스트 코드는 나중에 코드를 수정하더라도 두려움없이 코드 수정이 가능하도록 자동화가 가능

※ 화이트박스 테스트

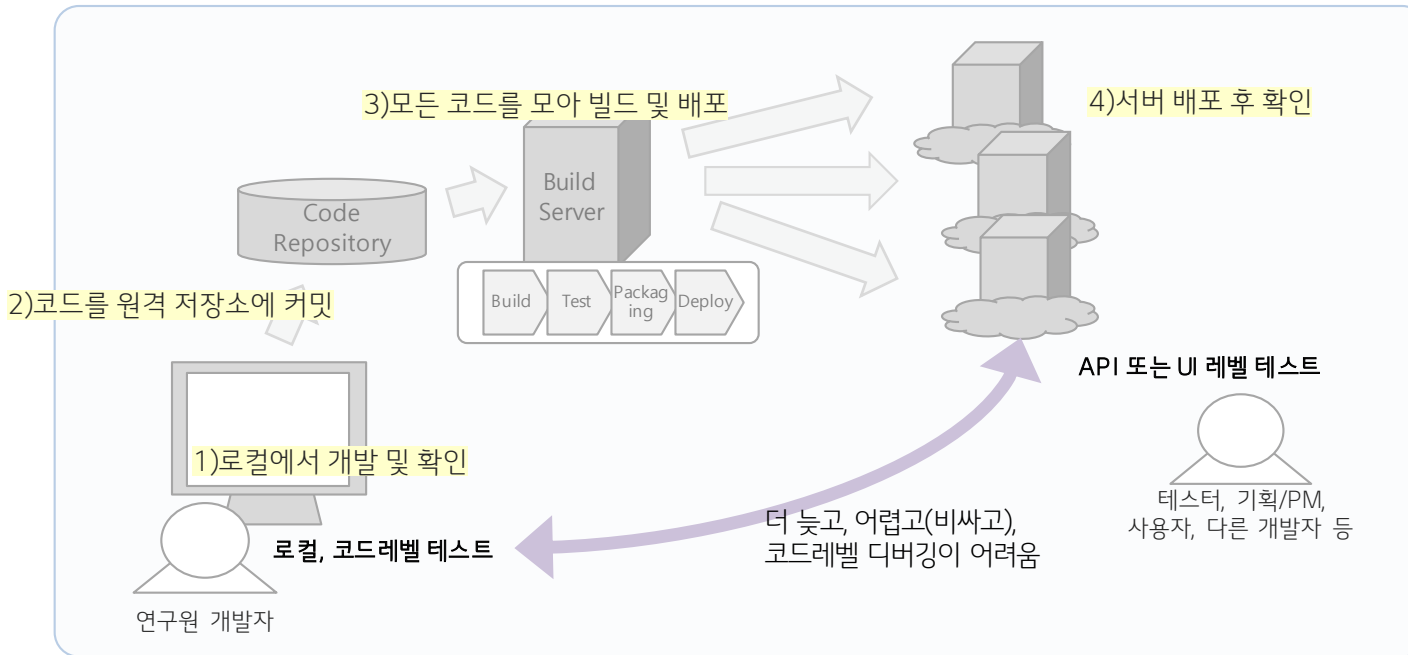
소프트웨어 혹은 제품의 내부 구조, 동작을 세밀하게 검사하는 테스트 방식으로, 외부에서 요구사항에 따른 예상 결과값을 테스트 하는 것과는 다르게 내부 소스 코드를 테스트하는 기법으로 외부에서는 볼 수 없는 코드 단위를 테스트 한다

즉, 정리하면 개발자가 소프트웨어 또는 컴포넌트 등의 로직에 대한 테스트를 수행하기 위해 설계 단계에서 요구된 사항을 확인하는 개발자 관점의 단위 테스트 기법이다



<https://catsbi.oopy.io/7c084479-c9d0-44a1-acb9-f6b43a19e332>
<https://www.professionalqa.com/white-box-testing>

코드 레벨(화이트 박스) 테스트의 필요성



- 개발 코드에 대한 테스트 코드 작성 및 수행(화이트 박스 테스트)
- 개발과 함께 바로, 빠르게 기능 동작을 확인(디버깅)
- 개발 IDE 내에서 쉽고 빠르게 수행 가능하며, 자동화가 쉬움
- 서버 배포 후 수행하는 상위 테스트 대비 수행 비용이 훨씬 저렴하고, 테스트 커버리지가 더 넓다
- 테스트 코드를 짜면서 코드 리팩토링이 되기도 하고, 나중에 코드를 수정하더라도 두려움없이 코드 수정이 가능

코드 레벨(화이트 박스) 테스트의 필요성

연구원에서도 화이트박스, 코드 기능 테스트가 필요할까? 중요할까?

연구만 잘하면 됐지,
코드 동작까지??...

테스트 코드를 따로 더 짜라고?
그 시간이면 개발코드를
더 짜고 말겠다!!!!

글쎄,...
뭐가 어떻다는 건지
와닿지 않는데?



아, 작성한 코드가 잘 동작하니,
연구에 더 집중할 수 있구나!

아, 테스트 코드를 짜면서 개발 코드가
정리되고(리팩토링),
한번 짰 테스트코드는 계속 돌리면서
확인할 수 있으니 이득이네!!

가이드의 샘플코드와 사례들을 참
고하니 뭔가 생각이 들기도 하네

※ “화이트박스,단위 테스트” vs “API,통합 테스트” 비교

이른 vs 늦은 테스트, 독립적인 vs 통합적인 테스트

- 개발과 동시에 디버그와 테스트가 가능하고, 이를 빌드 프로세스에 녹여 넣을 수 있음
- GUI 테스트 등 상위 테스트는 의존성과 복잡성이 높아져서 테스트를 할 수 있는 시기가 늦어지고, 테스트에 들어가는 공수가 증가
- 결함 발생 시 원인 파악, 디버그가 어려움
- 테스트 자동화 구축이 가장 용이하고, 이를 통해 코드 변경에 따른 걱정을 덜 수 있음

[하위 vs 상위 레벨 테스트]



[예: REST API 테스트와 코드레벨 테스트 비교]

항목	REST API 테스트	서버 내부 테스트
설명	노출된 URI(End-point)에 대해 정의된 스펙 기반으로 REST API 호출하여 테스트 수행	개발 소스 Commit 전에 개발 IDE 상에서 JUnit을 이용해 디버깅&단위테스트 용 테스트를 수행한다
수행목적	<ul style="list-style-type: none"> 클라이언트(앱) 입장에서 서버의 최종 인터페이스인 REST API에 대한 스펙 기반 상세 검증 자동/반복 테스트를 수행하며 스펙대로 동작하는지 상시 검증한다 향후 대상 IP (동적)변경으로 개발서버/스테이징 서버/ 운영 서버 배포 후 검증에 활용한다 * API Gateway를 통한 최종 API 호출을 검증한다 	<ul style="list-style-type: none"> 개발 소스 Commit 전에 개발한 기능 검증 용 개발환경에서 쉽고, 빠르게 기능 검증 코드 레벨 디버깅 테스트 커버리지 측정이 가능하다
주수행자	별도인력 / 서버 개발자	서버 개발자
수행방법	API 호출/검증 툴	JUnit (+spring-test)
수행시기	개발완료/소스 커밋/(개발/테스트)서버 반영 후	개발과 같은 스프린트
테스트 케이스 Depth	스펙 기반의 상세한 테스트 케이스 및 인증/접근 권한 테스트 등의 테스트 추가 수행	개발 및 디버깅 용도 수준의 depth

Pytest 란?

- Python 언어의 화이트박스 테스트 도구(프레임워크)
- 대표 사이트: <https://docs.pytest.org/>
- pytest의 장점
 - pytest는 구문이 단순하여 시작하기가 매우 쉽습니다
 - Fixture 등을 통해 미리 준비해 놓은 리소스 또는 코드 공유와 재사용을 지원합니다
 - pytest를 사용하면 실행 중에 테스트의 하위 집합을 건너뛰거나 특정 집합만을 실행할 수 있습니다(마커, 스킵 기능)

※ python 패키지에 기본으로 포함된 "unittest" 와의 단순 비교

(a) unittest: 더 오래되었다. xUnit 개념을 충실히 지키고 있다. 테스트를 위한 클래스를 정의하여 사용해야 한다

(b) pytest: 상대적으로 더 최근이고 더 널리 사용된다. 구문이 단순하고 쉽다

```
class calculator_machine:
```

```
    """
```

개발코드 예: 2개 값을 받아 덧셈 결과를 반환

```
    def sum(self,a,b):
```

```
        """
```

```
        if not isinstance(a,int) or type(b) is not int :  
            raise CalculateException("only int type is allowed")
```

```
        if a<0 or b<0:
```

```
            raise CalculateException("only more than zero value can input")
```

```
        return a+b
```

```
def test_sum_더큰값(my_fixture):  
    target = calculator_machine()  
    a = 2  
    b = 7  
    assert 9 == target.sum(a,b)
```

pytest 테스트 코드

```
class CalculatorTest(unittest.TestCase):
```

```
    def setup(self):  
        print("setup")
```

```
    def teardown(self):  
        print("teardown")
```

```
    def test_sum_bigger_b(self):
```

```
        target = calculator_machine()
```

```
        a = 2
```

```
        b = 7
```

```
        self.assertEqual(9, target.sum(a,b))
```

unittest 테스트 코드

※ Pytest 강점

“unittest” 대비 pytest의 강점

- 테스트 작성 시 불필요한 상용구 사용이 더 적다 (Less Boilerplate)
- 더 보기 좋은 테스트 결과 출력 (Nicer Output)
- 배우기 더 쉽다 (Less to Learn)
- Fixture를 통해 상태 및 종속성 관리 용이 (Easier to Manage State and Dependencies)
- 원하는/원하지 않는 테스트에 대한 필터링이 용이하다 (Easy to Filter Tests)
- 테스트 매개변수화 허용 (Allows Test Parametrization)
- 플러그인 기반 아키텍처 보유 (Has a Plugin-Based Architecture)
(pytest-randomly pytest-cov pytest-django pytest-bdd 등등)

<https://realpython.com/pytest-python-testing/>

1. 개요

2. Pytest 기본 사용법

3. Mocking을 통한 단위 테스트

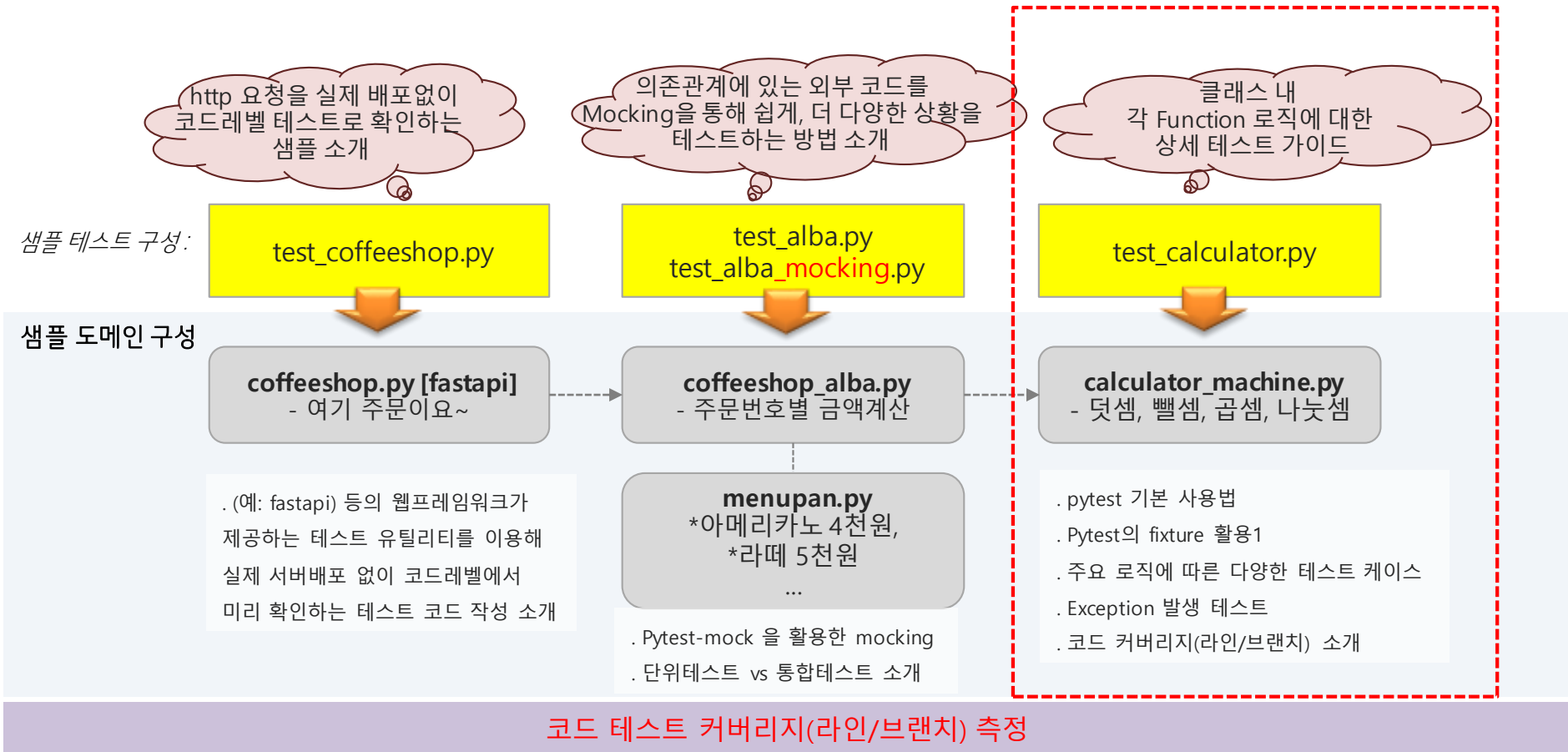
4. 코드레벨 통합 테스트(HTTP API 테스트)

5. 정리

가이드 구성 - 목적/상황별

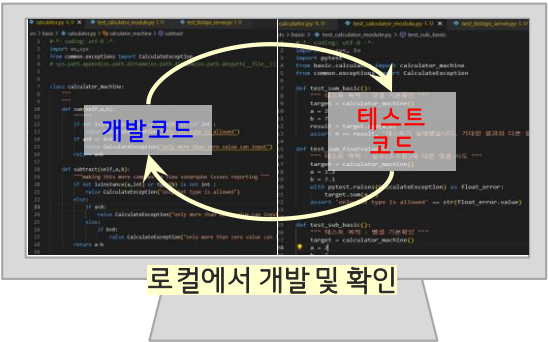
[샘플/가이드 구성안 - 초안]

- 계산기(AI 모델 모듈)를 이용해서 알바생(플랫폼 로직코드)이 커피 값을 계산하는 코드를 http(XXX서버)로 제공하는 서비스에 대해
- 각각 (a)알고리즘 코드 자체에 대한 테스트, (b,b')참조하는 모듈을 Mocking하며 하는 테스트, (c)실제 서버 배포 전에 코드레벨에서 http 요청을 확인하는 테스트 샘플을 제공



pytest 기본 사용법

단순 사칙연산 개발코드와 이에 대한 pytest 테스트 코드



개발코드 예

테스트 코드 예

덧셈(int a, int b)

- 2개 양수 간의 덧셈
- 1개 양수, 1개 0
- 1개 음수, 1개 양수
- 실수 값에 대한 덧셈 시도

.....

.....

```
6 class calculator_machine:
7     """
8     """
9     def sum(self,a,b):
10         """
11         """
12         if not isinstance(a,int) or type(b) is not int :
13             raise CalculateException("only int type is allowed")
14         if a<0 or b<0:
15             raise CalculateException("only more than zero value can input")
16         return a+b
17
18     def subtract(self,a,b):
19         """making this more complex to view sonarqube issues reporting """
20         if not isinstance(a,int) or type(b) is not int :
21             raise CalculateException("only int type is allowed")
22         else:
23             if a<0:
24                 raise CalculateException("only more than zero value can input")
25             else:
26                 if b<0:
27                     raise CalculateException("only more than zero value can input")
28         return a-b
29
30     def multiply(self,a,b):
```

개발코드

```
8
9 def test_sum_basic():
10     """ 테스트 목적 : 덧셈 기본확인 """
11     target = calculator_machine()
12     a = 2
13     b = 7
14     result = target.sum(a,b)
15     assert 9 == result, "테스트가 실패했습니다. 기대한 결과와 다른 결과 값이 나왔습니다 - " + str(result)
16
17 def test_sum_floatvalue():
18     """ 테스트 목적 : 실수(소수점)에 대한 덧셈 시도 """
19     target = calculator_machine()
20     a = 2.2
21     b = 7.1
22     with pytest.raises(CalculateException) as float_error:
23         target.sum(a,b)
24     assert 'only int type is allowed' == str(float_error.value)
25
26
27 def test_sub_basic():
```

테스트 코드

샘플1. 단순 덧셈, 뺄셈 등 코드에 대한 pytest 기본 사용법

1) pytest를 쓰려면 뭘 어떻게 해야 하나요?

2) 작성한 테스트를 실행시키고 싶어요,
디버깅하고 싶어요

3) 테스트 이름은 뭘로 지을까요?
어떤 폴더에 넣을까요?

4) 테스트 결과가 맞는지 매번 눈으로 확인하나요?

5) 예외 상황을 테스트하고 싶어요



연구원 개발자

6) Fixture, conftest.py는 뭔가요?

7) 반복적으로 여러 번 테스트를 해야 해요

8) 특정 테스트를 skip하고 싶어요,
특정 테스트만 실행시키고 싶어요

9) 테스트 데이터가 자꾸 바뀌어서 관리하기 힘들어요

10) 테스트를 얼마나 했는지 알고 싶어요

※ 샘플 프로젝트 준비

[준비물(교육 전)]

- 개인 노트북
- 개발IDE : VS Code
- Python 3.x 설치

[샘플 프로젝트 내려받기]

<https://gitlab.com/genycho/pytest-sample> 에서 파일 압축 다운로드(zip, tar.gz, tar,...) 후 압축해제 또는 Clone?

pytest-sample
Project ID: 47155195

7 Commits 1 Branch 0 Tags 1.9 MiB Project Storage

mocking 테스트 오류 수정
sangin chong authored 5 days ago

main pytest-sample

Download source code

zip tar.gz tar.bz2 tar

Name	Last commit	Last update
.vscode	first	3 weeks ago
src	mocking 테스트 오류 수정	5 days ago

(계정 체크 없음-업무 관련 코드는 없음)

[VS Code에서 폴더 Open]

- VS Code에서 폴더 Open
- VS Code 터미널에서 venv 생성 : `$ python -m venv ./venv`
- VS Code python 선택 : CTRL + Shift + 'P' 선택 > python : Select Interpreter 선택 > 생성한 venv의 python 선택
- venv 터미널 오픈 후 관련 라이브러리 설치 : `$ python -m pip install -r requirements.txt`

1) pytest를 쓰려면 뭘 어떻게 해야 하나요?

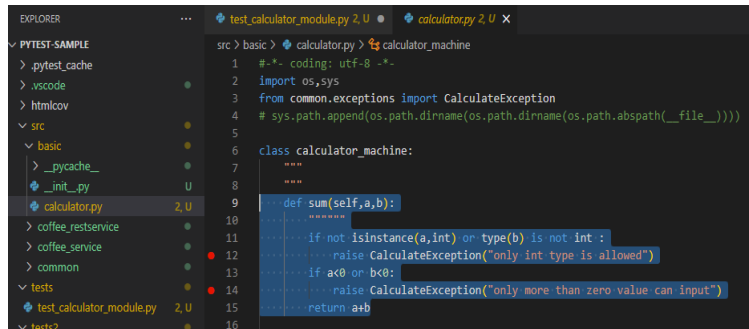
“pytest를 쓰려면 뭘 어떻게 해야 하나요?”

1) Pytest 설치

```
$ pip install pytest
```

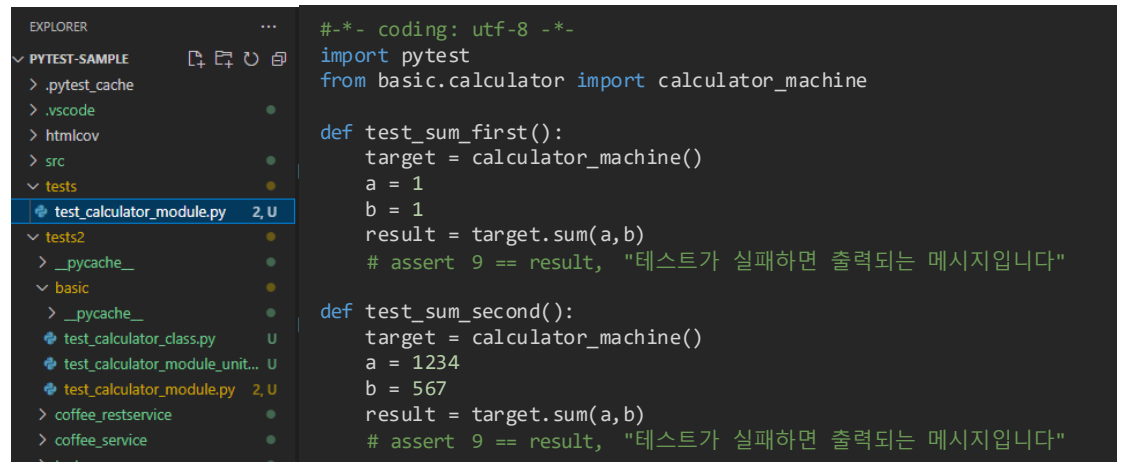
2) Pytest 테스트 파일을 생성합니다

0) 개발코드가 src/basic 폴더 하위 calculator.py에 작성되어 있다고 하면,



```
1  # -*- coding: utf-8 -*-
2  import os, sys
3  from common.exceptions import CalculateException
4  # sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
5
6  class calculator_machine:
7      """
8      """
9      def sum(self, a, b):
10         """
11         """
12         if not isinstance(a, int) or type(b) is not int:
13             raise CalculateException("only int type is allowed")
14         if a <= 0 or b <= 0:
15             raise CalculateException("only more than zero value can input")
16         return a + b
```

1) tests/basic 폴더 하위에 test_calculator.py 파일을 생성합니다



```
# -*- coding: utf-8 -*-
import pytest
from basic.calculator import calculator_machine

def test_sum_first():
    target = calculator_machine()
    a = 1
    b = 1
    result = target.sum(a, b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"

def test_sum_second():
    target = calculator_machine()
    a = 1234
    b = 567
    result = target.sum(a, b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"
```

1) pytest를 쓰려면 뭘 어떻게 해야 하나요?

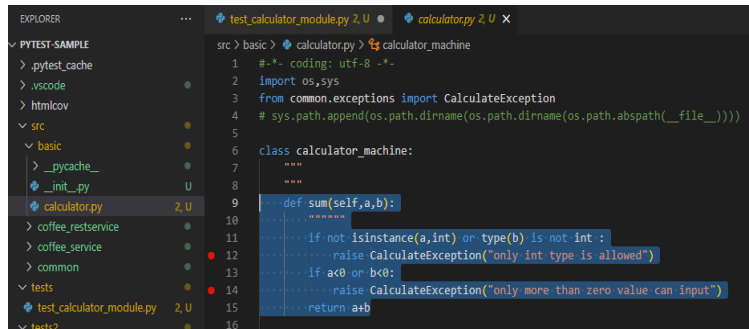
“pytest를 쓰려면 뭘 어떻게 해야 하나요?”

1) Pytest 설치

```
$ pip install pytest
```

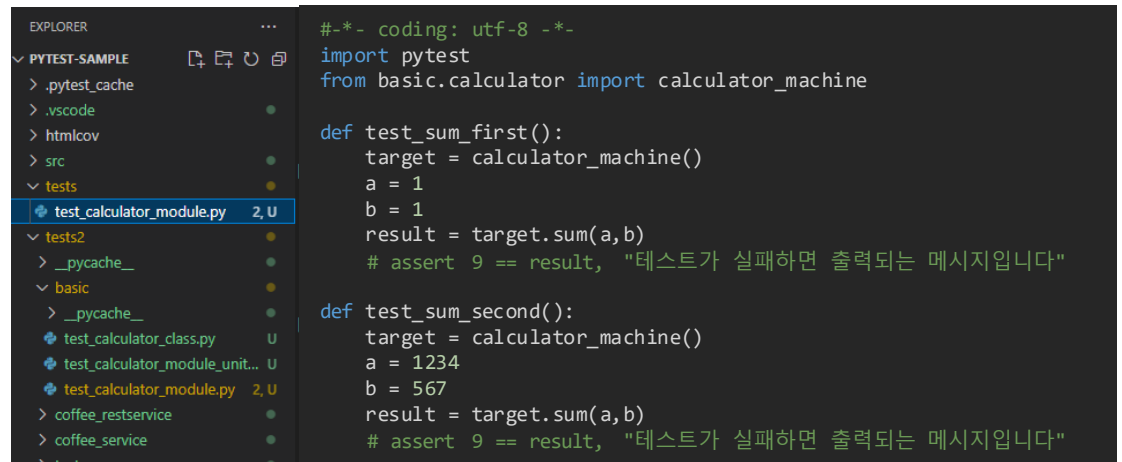
2) Pytest 테스트 파일을 생성합니다

0) 개발코드가 src/basic 폴더 하위 calculator.py에 작성되어 있다고 하면,



```
1  # -*- coding: utf-8 -*-
2  import os,sys
3  from common.exceptions import CalculateException
4  # sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
5
6  class calculator_machine:
7      """
8      """
9      def sum(self,a,b):
10         """
11         """
12         if not isinstance(a,int) or type(b) is not int :
13             raise CalculateException("only int type is allowed")
14         if a<0 or b<0:
15             raise CalculateException("only more than zero value can input")
16         return a+b
```

1) tests/basic 폴더 하위에 test_calculator.py 파일을 생성합니다



```
# -*- coding: utf-8 -*-
import pytest
from basic.calculator import calculator_machine

def test_sum_first():
    target = calculator_machine()
    a = 1
    b = 1
    result = target.sum(a,b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"

def test_sum_second():
    target = calculator_machine()
    a = 1234
    b = 567
    result = target.sum(a,b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"
```


1) pytest를 쓰려면 뭘 어떻게 해야 하나요?

3) Pytest 테스트 코드 작성

```
#!/usr/bin/env python3
#-*- coding: utf-8
import pytest
from basic.calculator import calculator_machine

def test_sum_first():
    target = calculator_machine()
    a = 1
    b = 1
    result = target.sum(a,b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"

def test_sum_second():
    target = calculator_machine()
    a = 1234
    b = 567
    result = target.sum(a,b)
    # assert 9 == result, "테스트가 실패하면 출력되는 메시지입니다"
```

2) pytest를 impor하고

3) 테스트 함수(케이스)를 test_sum_first 이름으로 작성한다

3-a) 테스트 대상 클래스를 생성

3-b) 테스트 데이터를 정의하고, 실행

3-c) 테스트 결과를 검증합니다

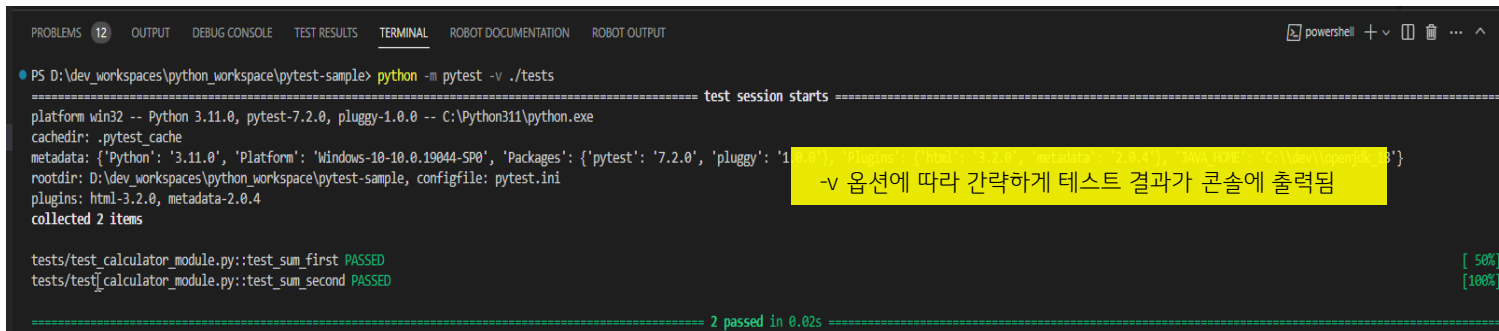
4) 1개의 함수에 대해 여러 테스트 함수(케이스)를 추가

2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

(VS Code) 작성한 테스트 실행, 디버깅 방법

[커맨드 라인에서 실행하기]

```
$ python -m pytest -v ./tests
```



```
PS D:\dev_workspaces\python_workspace\pytest-sample> python -m pytest -v ./tests

===== test session starts =====
platform win32 -- Python 3.11.0, pytest-7.2.0, pluggy-1.0.0 -- C:\Python311\python.exe
cachedir: .pytest_cache
metadata: {'Python': '3.11.0', 'Platform': 'Windows-10-10.0.19044-SP0', 'Packages': {'pytest': '7.2.0', 'pluggy': '1.0.0'}, 'Plugins': {'html': '3.2.0', 'metadata': '2.0.4'}}
rootdir: D:\dev_workspaces\python_workspace\pytest-sample, configfile: pytest.ini
plugins: html-3.2.0, metadata-2.0.4
collected 2 items

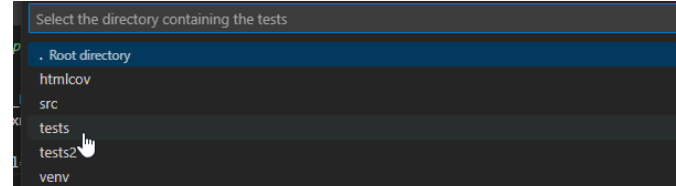
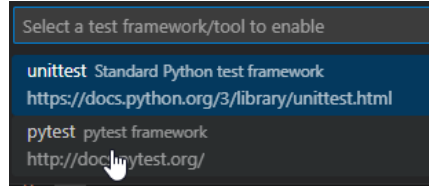
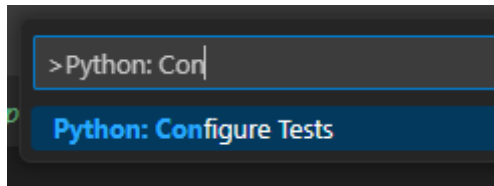
tests/test_calculator_module.py::test_sum_first PASSED [ 50%]
tests/test_calculator_module.py::test_sum_second PASSED [100%]

===== 2 passed in 0.02s =====
```

-v 옵션에 따라 간략하게 테스트 결과가 콘솔에 출력됨

[VS Code에서 pytest 실행하기]

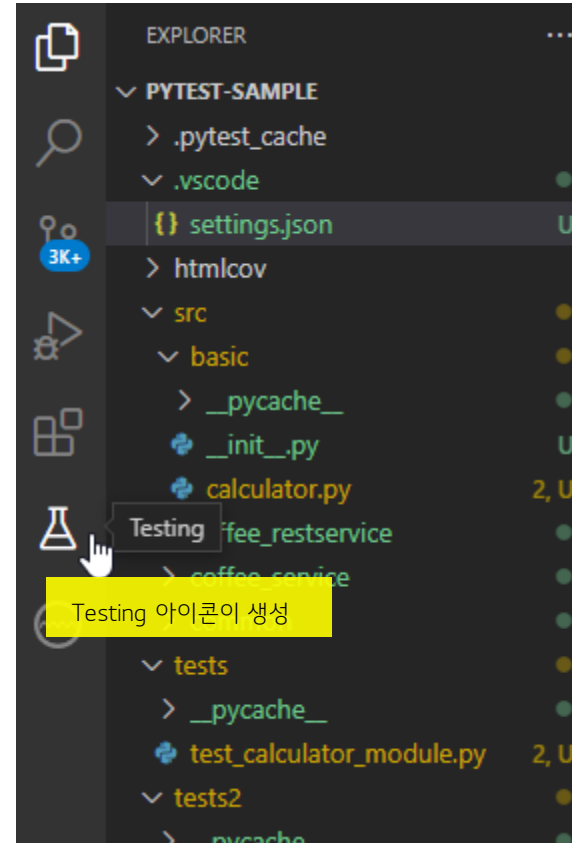
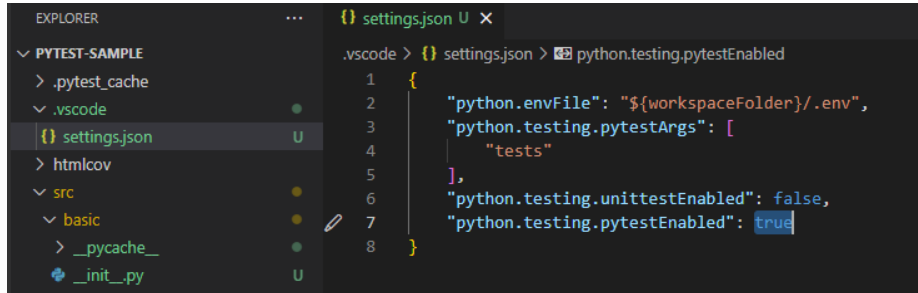
- 1) Ctrl + Shift + 'p' (또는 상단 View>Command Palette 선택)를 선택한 후 “Python: Configure Tests” 를 선택합니다
- 2) 해당 프로젝트의 테스트를 “pytest”로 선택합니다
- 3) 테스트 코드가 존재하는 디렉토리를 앞에서 생성한 “tests” 디렉토리로 선택합니다



2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

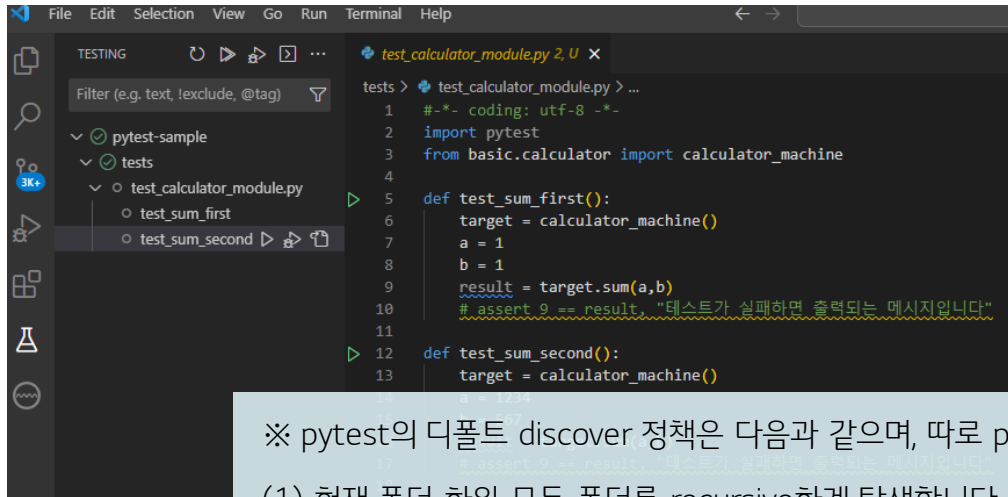
4) Pytest 설정이 끝나면 설정 내용이 .vscode/settings.json에 반영되며, vs code의 좌측에 “Testing” 메뉴가 표시됩니다



2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

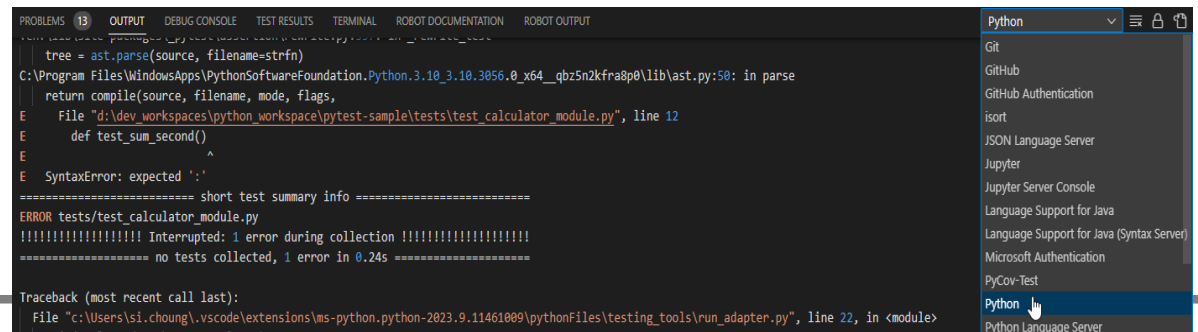
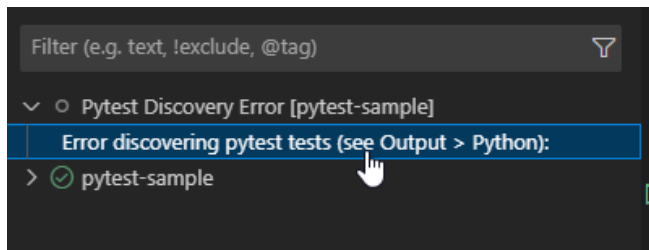
5) 좌측의 “Testing” 아이콘을 선택합니다 . Pytest는 테스트 코드를 내부적으로 자동으로 discover한 후 발견한 테스트 코드들을 표시해 줍니다.



※ pytest의 디폴트 discover 정책은 다음과 같으며, 따로 pytest.ini 파일을 생성하여 커스터마이징할 수 있습니다.

- (1) 현재 폴더 하위 모든 폴더를 recursive하게 탐색합니다 (또는 지정한 tests 폴더 하위를 탐색)
- (2) test_로 시작하거나 _test로 끝나는 파일을 탐색합니다 (파일명은 test_*.py 또는 *_test.py)
- (3) test로 시작하는 함수를 탐색합니다

5') “Pytest Discovery Error”가 나서 테스트가 표시 안 되는 경우에는 Output>Python 내용을 확인하여 수정하고 Refresh 합니다



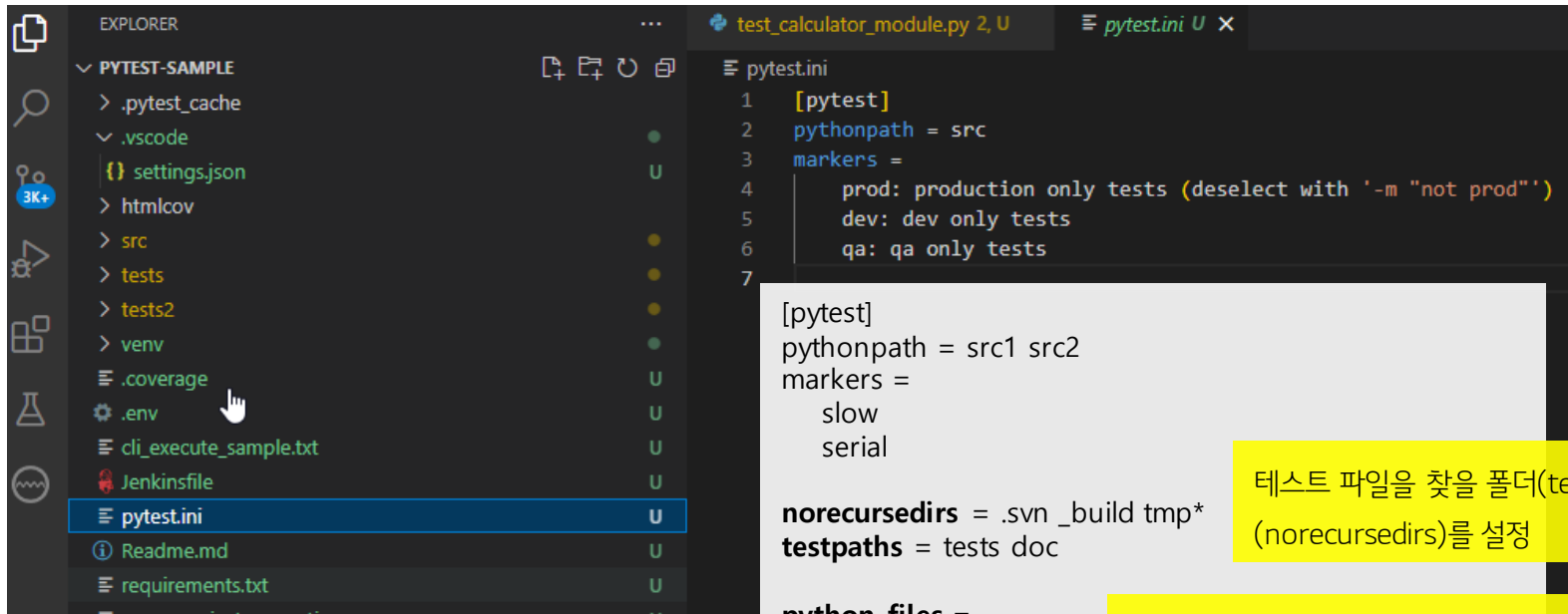
2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

※ 따로 `pytest.ini` 파일로 `discovery` 설정 커스터마이징하기

(참고) `pytest.ini` 설명 : <https://docs.pytest.org/en/latest/reference/reference.html#configuration-options>

1) 프로젝트 폴더 하위에 `pytest.ini` 파일을 생성하고, 설정을 커스터마이징 할 수 있습니다



```
[pytest]
pythonpath = src1 src2
markers =
    slow
    serial
```

```
norecursedirs = .svn _build tmp*
testpaths = tests doc
```

```
python_files =
    test_*.py
    check_*.py
    example_*.py
```

```
python_functions = *_test
```

테스트 파일을 찾을 폴더(testpaths)나 찾지 않을 폴더(norecursedirs)를 설정

테스트 파일 패턴을 python_files 에서 설정

테스트 함수 패턴을 python_functions 에서 설정

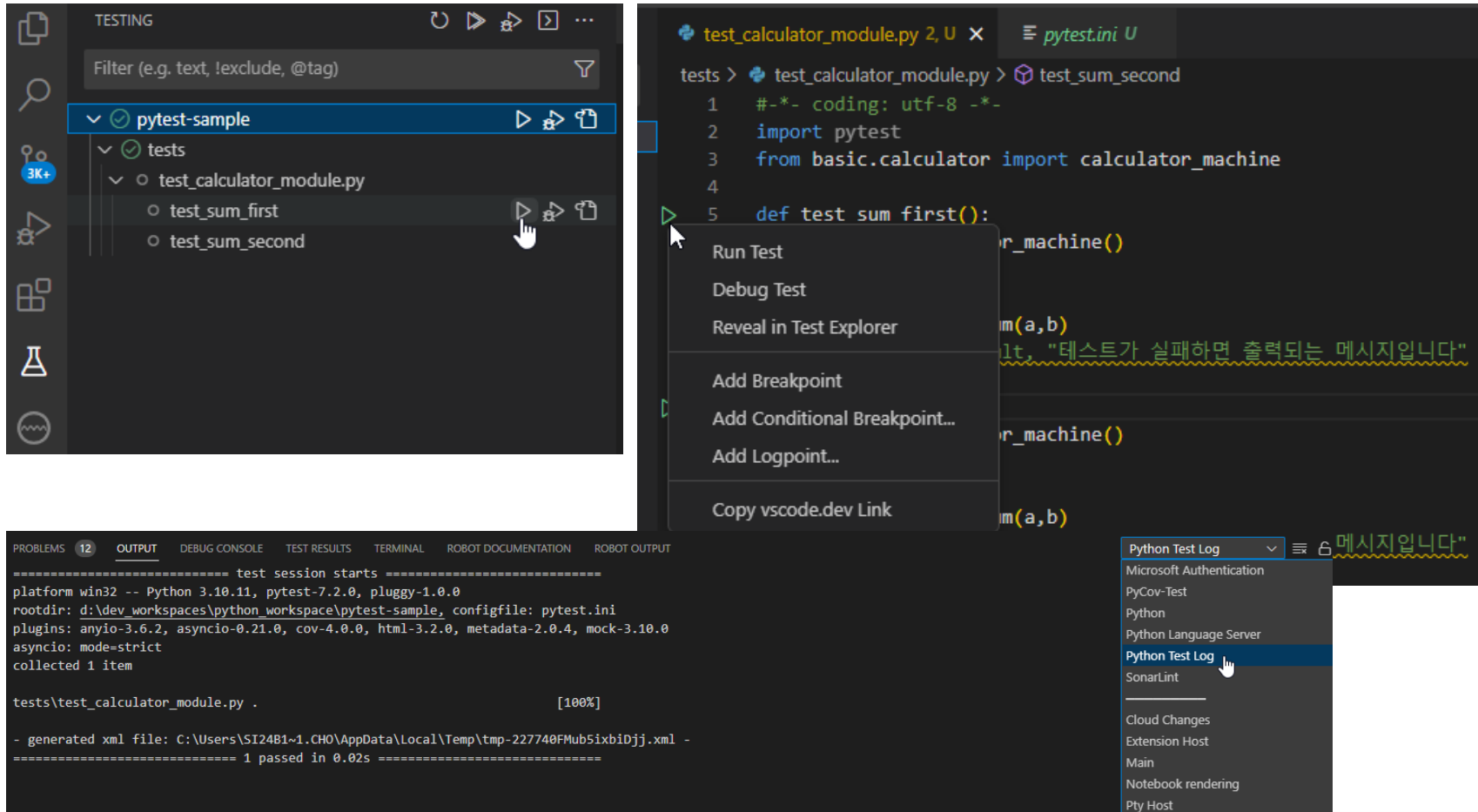
2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

6-a) Testing 패널에 표시된 테스트에 대해 Run Test 하거나

6-b) VS Code의 편집기에서 각 함수별로 Run Test 를 선택하여 테스트 실행이 가능합니다

테스트 로그는 Output > Python Test Log에서 확인 가능합니다



2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

6-c) 테스트가 실패하면 실패 내용이 표시됩니다

The screenshot displays the VS Code interface during a pytest run. On the left, the 'TESTING' sidebar shows a tree view where 'test_calculator_module.py' is expanded, and 'test_sum_first' is marked as failed. The main editor shows the source code of 'test_calculator_module.py' and 'calculator_machine.py'. The 'test_sum_first' function is highlighted with a red error message. The bottom panel shows the 'OUTPUT' window with the detailed error message.

```
tests > test_calculator_module.py > test_sum_first
1  -*- coding: utf-8 -*-
2  import pytest
3  from basic.calculator import calculator_machine
4
5  def test_sum_first():
    target = calculator_machine()
    a = 1
    b = -1
    result = target.sum(a,b)

tests\test_calculator_module.py:9:
-----
self = <basic.calculator.calculator_machine object at 0x0000016875BA1FC0>, a = 1
b = -1

def sum(self,a,b):
    """
    if not isinstance(a,int) or type(b) is not int :
        raise CalculateException("only int type is allowed")
    if a<0 or b<0:
        raise CalculateException("only more than zero value can input")
    common.exceptions.CalculateException: only more than zero value can input
>
E
6  target = calculator_machine()
>
> result = target.sum(a,b)

tests\test_calculator_module.py:9:
-----
self = <basic.calculator.calculator_machine object at 0x0000016875BA1FC0>, a = 1
b = -1

def sum(self,a,b):
    """
    if not isinstance(a,int) or type(b) is not int :
        raise CalculateException("only int type is allowed")
    if a<0 or b<0:
        raise CalculateException("only more than zero value can input")
E
common.exceptions.CalculateException: only more than zero value can input

src\basic\calculator.py:14: CalculateException
- generated xml file: C:\Users\SI24B1~1\CHO\AppData\Local\Temp\tmp-227740395qQgdInZ29.xml -
```

Testing 패널에 실패 표시

각 테스트 함수 선언부분에 실패 내용 표시

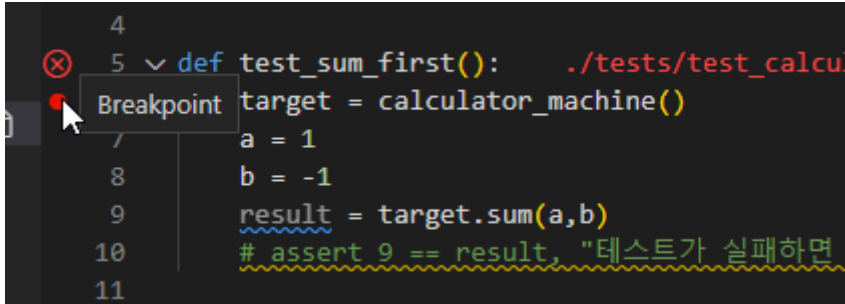
OUTPUT> Python Test Log에 상세 내용 출력

2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

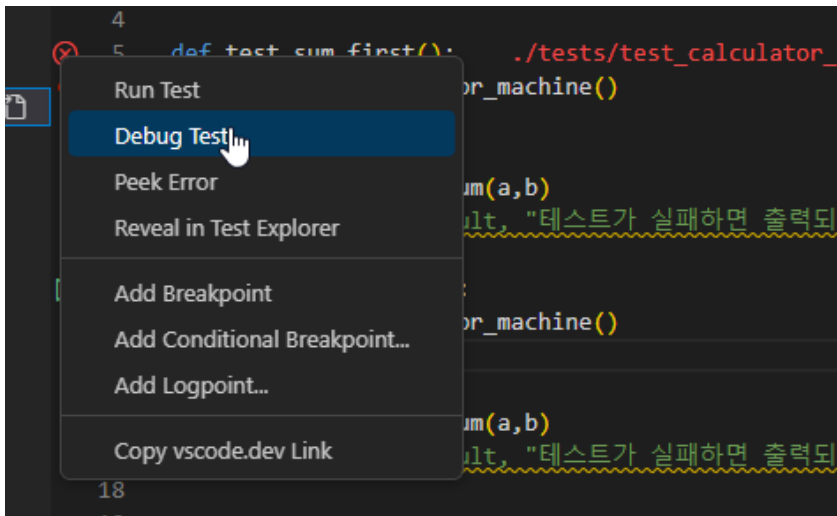
[VS Code에서 pytest 실행하기]

테스트가 실패한 이유를 찾기 위해 디버깅(한 줄씩 수행하며, 그때그때 값 변화를 확인) 을 해 볼 수 있습니다

7-a) VS Code의 에디터 창에 테스트 코드의 왼쪽 라인 영역을 클릭하여 break-point를 찍습니다



7-b) 테스트 코드를 Run 이 아닌 Debug로 실행합니다



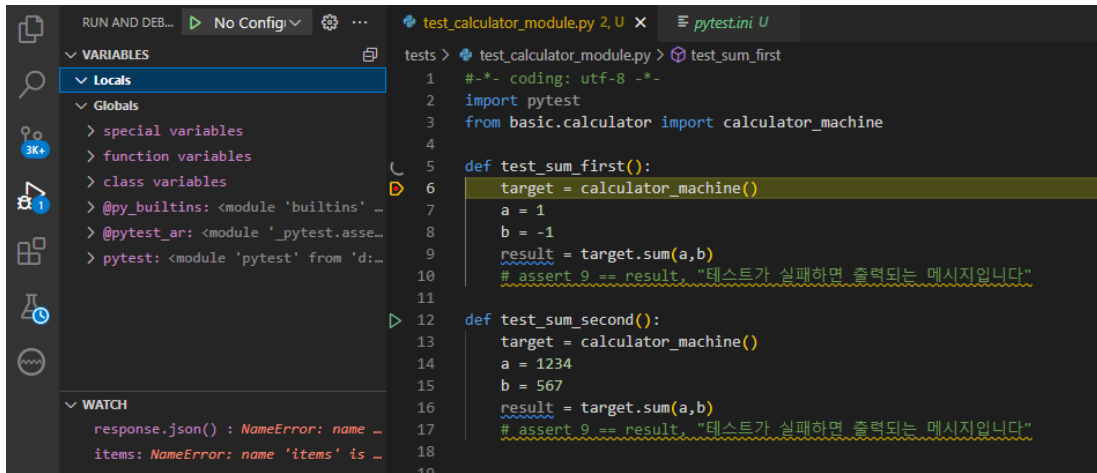
2) 작성한 테스트를 실행시키고 싶어요, 디버깅하고 싶어요

[VS Code에서 pytest 실행하기]

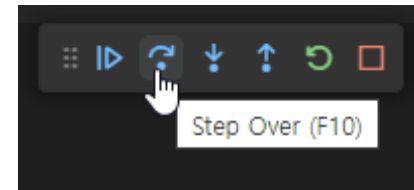
7-c) 이전에 찍은 break-point에서 프로그램 실행이 일시 정지되며, 디버그 실행 명령어를 이용해 1라인씩 실행합니다

(*) 이때, 좌측의 "VARIABLES"탭의 변수 값들을 참고합니다.

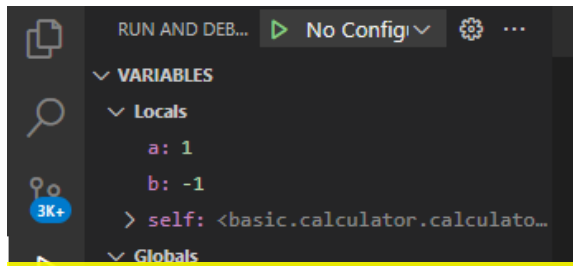
(*) 필요한 경우 WATCH 탭에 확인하고 싶은 변수 명, 코드 식을 입력하여 현재 시점의 값을 확인할 수 있습니다



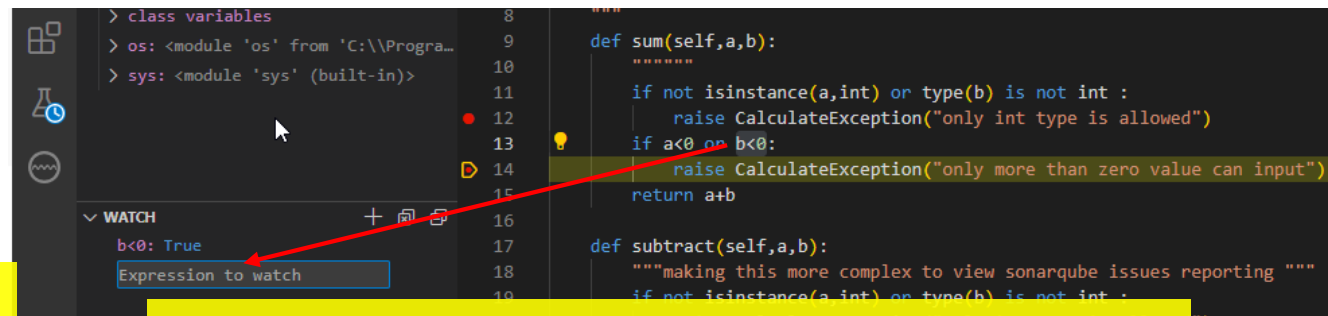
[디버그 실행 명령어]



- Continue (F5) : 다음 break-point까지 바로 실행
- Step Over(F10) : 현재 함수에서 1라인씩 실행
- Step Into(F11) : 현재 함수가 호출하는 다른 함수로 진입
(*테스트 코드에서 개발 코드로 진입할 때 사용)
- Step Out(Shift F11) : 현재 함수를 호출한 상위 함수로 바로 이동
- Restart : 현재 테스트를 끝까지 실행/종료 시킨 후 다시 실행
- Stop : 현재 테스트 종료



VARIABLES 탭에서 해당 시점의 변수 값들을 확인



개발코드의 에러가 난 부분에서

WATCH 탭에 if 조건 식을 넣어보고 에러가 발생한 상세 원인을 확인

3) 테스트 이름은 뭘로 지을까요? 어떤 폴더에 넣을까요?

테스트 이름은 뭘로 지을까요? 어떤 폴더에 넣을까요?

[일반적으로 사용되는 테스트 이름(파일, 함수명 등)]

테스트 이름 대상	이름 명명 규칙	예
테스트 파일명	test_{대상}.py	test_calculator.py
테스트 함수명 1	test_{대상}_{테스트목적}	test_sum_morethanmaxvalue
테스트 함수명 2	test_{대상}_{입력값}_{기대값}	test_sum_양수값_정상덧셈결과

[일반적인 테스트 코드(파일) 관리 방식 2가지]

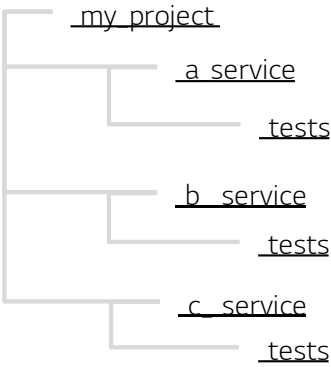
코드 배포 시 테스트 코드는 배포되지 않도록 개발 코드와 구분/분리가 필요하며, 일반적으로 별도 테스트 디렉토리를 지정하여 관리 함

1번 방식) 프로젝트 상위에서 별도 디렉토리(tests)를 지정하여 분리

[pytest recommends](#) including an additional directory to separate the source code within a project:

```
my_package
├── src # <-- no __init__.py on this layer
│   └── my_package
│       ├── __init__.py
│       ├── util_module
│       │   ├── __init__.py
│       │   └── utils.py
└── tests
    ├── __init__.py
    ├── test_util_module
    │   ├── __init__.py
    │   └── test_utils.py
```

2번 방식) 각 개발코드 디렉토리 하위에 별도 tests 디렉토리를 생성하여 분리



4) 테스트 결과가 맞는지 매번 눈으로 확인하나요?

테스트 결과 확인하는 assert 문

pytest 가 지원하는 “assert”를 통해 기대 값과 실제 결과 값이 일치하는지 확인하는 코드를 작성합니다

※ 일반적인 assert 상황

(1) 같거나 같지 않은지를 확인

- assert {기대 값} == {결과 값} # Success,
- assert {기대 값(나오면 안 되는 값)} != {결과 값} # Success,

(2) 타입, 인스턴스 확인

- assert type(5) is int
- assert isinstance('5', str)

(3) Boolean 결과 확인

- assert true == result, assert true is True

(4) in and not in [iterable]

- list_one=[1,3,5,6]
- assert 5 in list_one

(5) Greater than or less than [value]

- assert 5 > 4 # Success

[덧셈 결과가 기대 값과 일치하는지 확인하는 assert 문 추가]

```
#-*- coding: utf-8 -*-
import pytest
from basic.calculator import calculator_machine

def test_sum_first():
    target = calculator_machine()
    a = 1
    b = 1
    result = target.sum(a,b)
    assert 2 == result, "테스트가 실패하면 출력되는 메시지입니다"

def test_sum_second():
    target = calculator_machine()
    a = 1234
    b = 567
    result = target.sum(a,b)
    assert 0 == result
```

Assert 문 끝에 문장을 작성하면, 테스트 실패 시에 해당 문장이 같이 출력되어 테스트 실패 원인 파악이 쉬워진다

5) 예외 상황을 테스트하고 싶어요

Exception 발생을 확인하는 테스트

[Exception 발생을 확인하는 테스트]



개발자

응? 왜 에러(XXXException)가 발생 안 했는데,
테스트는 실패하죠?

저희가 일부러 에러 상황을 만들었을 때,
기대한 대로 의도한 특정 에러가 나는지
확인하는 테스트여서 그래요

웬, 그런 테스트(에러가 나는)는
대체 왜 하는거죠?

실제로도 발생할 수 있는 예외적인 상황에 대해
저희가 작성한 코드가 의도한대로 잘 대응하는지
확인하는 거죠. 화이트박스 테스트의 묘미죠!!



QA

```
with pytest.raises(CalculateException) as expect_exec:
    target.sum(a,b) #코드실행
assert 'exception message check!!' == str(expect_exec.value)
```

```
1  -*- coding: utf-8 -*-
2  import os,sys
3  from common.exceptions import CalculateException
4  # sys.path.append(os.path.dirname(os.pa
5
6
7  class calculator_machine:
8
9      .. ..
10
11      def sum(self,a,b):
12
13          if not isinstance(a,int) or type(b) is not int :
14              raise CalculateException("only int type is allowed")
15          if a<0 or b<0:
16              raise CalculateException("only more than zero value can input")
17          return a+b
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

자연수만 지원하는 계산기 프로그램에 실수나 음수
값이 들어왔을 때 '계산오류' - CalculateException
이 발생하게 되어 있는 개발 코드

```
17 def test_sum_floatvalue():
18     """ 테스트 목적 : 실수(소수점)에 대한 덧셈 시도 """
19     target = calculator_machine()
20     a = 2.2
21     b = 7.1
22     with pytest.raises(CalculateException) as float_error:
23         target.sum(a,b)
24     assert 'only int type is allowed' == str(float_error.value)
25
```

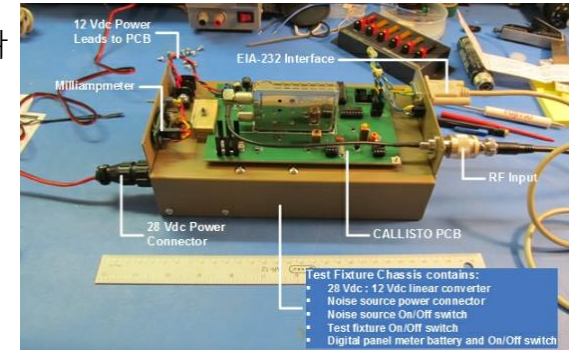
6) Fixture, conftest.py는 뭔가요?

pytest의 **fixture**와 fixture모음 **conftest.py**

[Fixture]

※ Test Fixture

- 테스트 픽스처는 일부 항목, 장치 또는 소프트웨어를 일관되게 테스트하는 데 사용되는 것들을 의미합니다
- 테스트 픽스처의 목적은 결과가 반복 가능하도록 테스트가 실행되는 고정된 환경을 제공하는 것입니다
- 테스트 픽스처를 사용하면 매번 동일한 설정으로 시작하므로 테스트를 반복할 수 있습니다
- 소프트웨어 테스트 픽스처의 예 :
 - 입력 데이터 준비 및 가짜 또는 모의 개체 설정/생성
 - 알려진 특정 데이터 세트로 데이터베이스 로드
 - 필요한 특정 파일 세트를 복사하여 특정 상태로 초기화된 객체를 준비



※ Pytest의 fixture

테스트 코드 내 중복되는 코드를 fixture로 줄여 간결하게 만들 수 있습니다

- 1) 동일하게 반복되는 코드를 하나의 함수로 빼서 fixture 를 선언하고, 다른 함수에서는 이 함수명을 인자로 받아와서 사용할 수 있다
- 2) 여러 번 사용(호출)할 수 있다. test3, test4, ... 등 테스트 함수를 추가로 선언하여 원할 때 언제든지 data 를 인자로 불러올 수 있다. 즉, fixture 로 선언된 함수는 reusable 하다
- 3) fixture 는 다른 fixture 를 호출할 수도 있다
- 4) 한 번에 여러 fixture 를 호출할 수도 있다

6) Fixture, conftest.py는 뭔가요?

[Fixture]

- 특정 함수에 @pytest.fixture를 선언한 후 다른 테스트에 함수 인자로 이 fixture를 주입할 수 있습니다 (여러 테스트 함수에서 재사용)
- 계산기(Calculator) 클래스를 생성한 후 초기화를 하는 fixture를 정의한 후 테스트 함수에서 사용합니다
- fixture에서 yield문으로 teardown 기능을 사용할 수 있습니다
- fixture 함수는 다른 fixture를 인자로 받을 수 있습니다
- 여러 개의 fixture를 인자로 받을 수 있습니다

```
@pytest.fixture(scope="function")
def get_calculator():
    calc = calculator_machine()
    calc.reset()
    # return calc
    yield calc
    # "테스트가 완료되면 yield문 이후 코드가 실행됩니다"
```

calculator_machine() 클래스를 생성한 후 반환하는 샘플 fixture를 작성합니다

```
def test_sum_basic(get_calculator):
    """ 테스트 목적 : 덧셈 기본확인 """
    target = get_calculator
    a = 2
    b = 7
    result = target.sum(a,b)
    assert 9 == result, "테스트 값이 나왔습니다 - " + str(result)
```

일반 테스트 함수에서는 fixture 함수 이름을 인자로 전달하면 fixture가 정해진 scope에 따라 실행 후 재사용할 수 있습니다

※ fixture scope 종류 및 적용

fixture는 테스트에서 처음 요청될 때 생성되며 해당 범위에 따라 소멸됩니다 (해당 scope 범위 내에서는 1회만 수행되고 재사용됩니다)

- **function**: 선언하지 않은 경우 **디폴트로 적용되는 scope**입니다. fixture는 테스트 종료 시 소멸됩니다
- **class**: 테스트 클래스의 마지막 테스트 종료 시 fixture가 소멸됩니다
- **module**: 모듈의 마지막 테스트 종료 시 fixture가 소멸됩니다
- **package**: 패키지의 마지막 테스트를 분해하는 동안 fixture가 소멸됩니다
- **session**: 테스트 세션이 끝나면 fixture가 소멸됩니다

6) Fixture, conftest.py는 뭔가요?

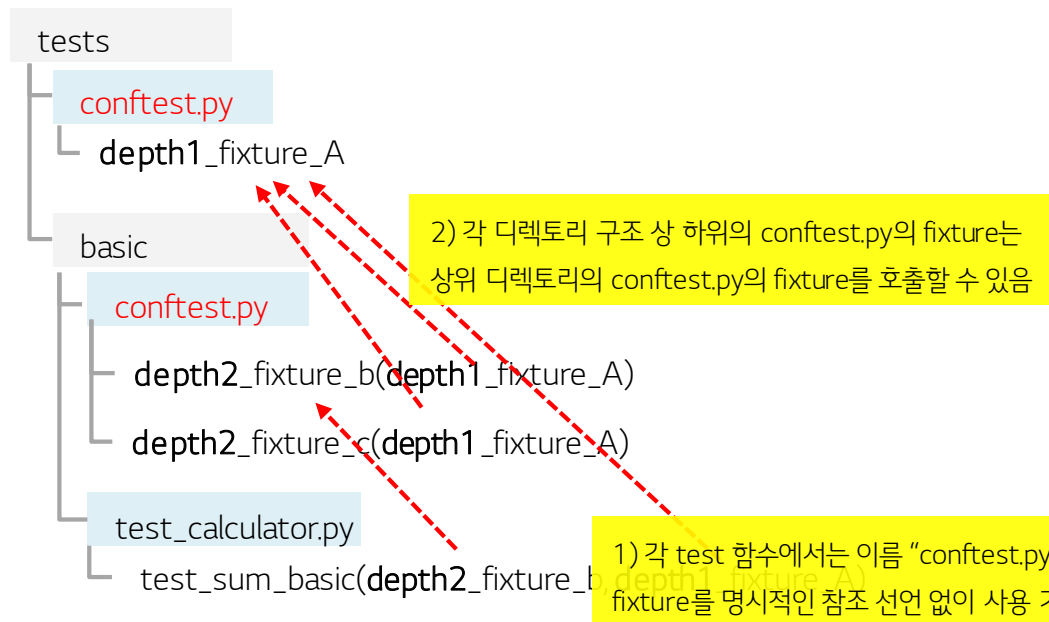
[conftest.py]

- 각 테스트에서 정의한 fixture들을 "conftest.py"라는 파일에 모아두면, 하위의 모든 pytest에서 명시적인 참조없이 재사용이 가능합니다 (pytest가 자동으로 검색)
- 또한, 이 conftest.py는 상위 디렉토리의 conftest.py 내용을 상속받는 형태로 동작합니다 (<https://docs.pytest.org/en/6.2.x/fixture.html>)

예를 들면,

- 1) 각 test 함수에서는 이름 "conftest.py"에 정의된 fixture를 명시적인 참조 선언 없이 사용 가능하며,
- 2) 각 디렉토리 구조 상 하위의 conftest.py의 fixture는 상위 디렉토리의 conftest.py의 fixture를 호출할 수 있음

[디렉토리 구조와 conftest.py]



[conftest.py 작성예]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import os, sys, io
import pytest
from basic.calculator import calculator_machine
from common.exceptions import CalculateException

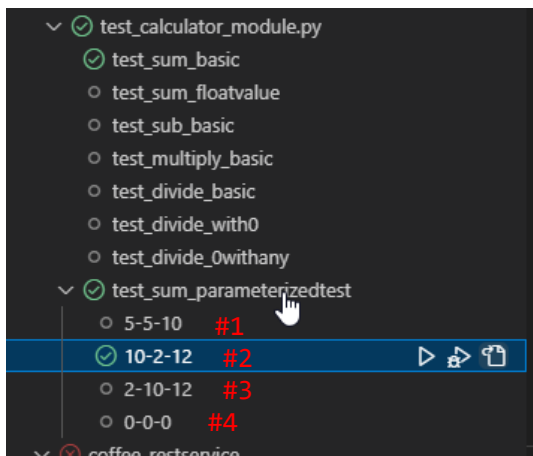
@pytest.fixture(scope="function")
def get_calculator():
    calc = calculator_machine()
    calc.reset()
    # return calc
    yield calc
    # "테스트가 완료되면 yield문 이후 코드가 실행됩니다"
```

7) 반복적으로 여러 번 테스트를 해야 해요

pytest의 parmeterization

[Parameterization 테스트]

- 1) 테스트 함수 상단에 `@pytest.mark.parametrize` 정의
- 2) 한 셋으로 여러 번 수행할 변수 명을 입력 (예) 2개의 숫자값을 입력 받은 후 그 결과를 확인하는 테스트에 대해 "input_a, input_b, expected" 라는 변수명을 정의
- 3) 테스트 함수의 fixture 입력에 위 변수 명을 지정 (예) (input_a, input_b, expected)
- 4) 이후 테스트 함수 내에서 fixture 사용과 동일하게 사용
- 5) 테스트를 실행시키면 준비한 셋만큼 반복적으로 실행되고, 특정 셋만 골라서 실행시킬 수도 있다



```
@pytest.mark.parametrize("input_a, input_b, expected", {(10, 2, 12), (2, 10, 12), (5, 5, 10), (0, 0, 0)})
def test_sum_parameterizedtest(input_a, input_b, expected):
    """ 테스트 목적 : 덧셈에 대한 parameterized 테스트 예 """
    target = calculator_machine()
    a = input_a
    b = input_b
    assert expected == target.sum(a, b)
```

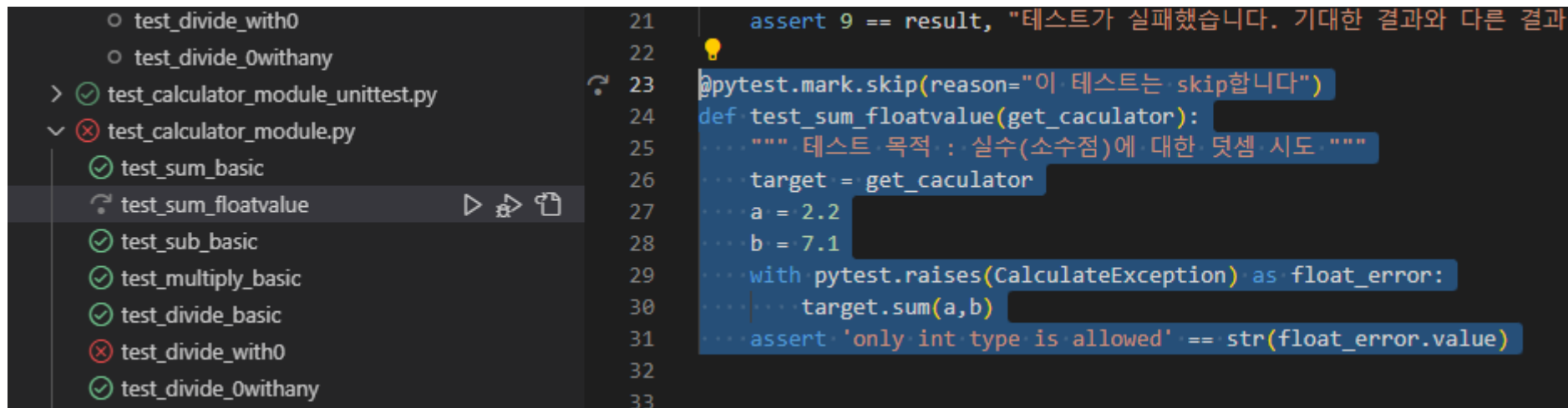
Diagram illustrating the parameterization process. Red arrows point from the parameter names `input_a`, `input_b`, and `expected` in the `@pytest.mark.parametrize` decorator to their corresponding arguments in the test function `test_sum_parameterizedtest`. Additionally, red arrows point from the parameter names to the specific values in the test cases: `(10, 2, 12)`, `(2, 10, 12)`, and `(5, 5, 10)`. The test cases are labeled #1, #2, and #3 respectively.

8) 특정 테스트를 skip하고 싶어요, 특정 테스트만 실행시키고 싶어요

pytest의 skip, marker

[pytest의 [skip](#)으로 특정 테스트를 아예 실행하지 않기]

1) 특정 테스트를 실행시키지 원하지 않는 경우 해당 테스트 함수 상단에 `@pytest.mark.skip(reason="skip시키려는 이유")` 를 작성하면 테스트 실행 시 skip되며, 작성한 reason 내용과 같이 구분되어 출력



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a test suite with several tests, some of which are skipped. The code editor shows a test function that is skipped.

```
21 assert 9 == result, "테스트가 실패했습니다. 기대한 결과와 다른 결과"
22
23 @pytest.mark.skip(reason="이 테스트는 skip합니다")
24 def test_sum_floatvalue(get_caculator):
25     """테스트 목적 : 실수(소수점)에 대한 덧셈 시도"""
26     target = get_caculator
27     a = 2.2
28     b = 7.1
29     with pytest.raises(CalculateException) as float_error:
30         target.sum(a,b)
31     assert 'only int type is allowed' == str(float_error.value)
32
33
```

8) 특정 테스트를 skip하고 싶어요, 특정 테스트만 실행시키고 싶어요

[pytest의 **marker**로 특정 테스트 그룹 별로 실행/미실행 설정]

(상황 예) 개발 환경에서만 수행하려는 테스트가 있는 경우

(상황 예) unit-testing(상세한 테스트)와 integration-testing(주요 흐름 확인)를 구분해서 상세한 테스트와 빠른 테스트로 구분하려는 경우

1) (선택) 미리 pytest.ini 파일에 marker를 정의

2) 테스트 함수 상단에 @pytest.mark.**marker** 작성

3) pytest 실행 시 -m 옵션을 이용하여 원하는 테스트만 실행 가능

1) (선택) 미리 pytest.ini 파일에 marker를 정의

```
@pytest.mark.prod
```

```
@pytest.mark.dev
```

```
def test_sub_basic():
```

```
    """ 테스트 """
```

```
    target = 10
```

```
    a = 2
```

```
    b = 7
```

```
    result = target.subtract(a,b)
```

```
    assert -5 == result
```

2) 테스트 함수 상단에 @pytest.mark.marker

```
$ pytest -v ./tests -m prod
```

```
$ pytest -v ./tests -m "not integration-testing"
```

not, and, or 키워드로 구성된 복합 수식 사용 가능

3) pytest 실행 시 -m 옵션을 이용하여 원하는 테스트만 실행 가능

```
(venv) PS D:\dev_workspaces\python_workspace\pytest-sample> pytest -v ./tests -m prod
===== test session starts =====
platform win32 -- Python 3.10.11, pytest-7.2.0, pluggy-1.0.0 -- D:\dev_workspaces\python_workspace\pytest-sample\venv\Scripts\python.exe
cachedir: .pytest_cache
metadata: {'Python': '3.10.11', 'Platform': 'Windows-10-10.0.19044-SP0', 'Packages': {'pytest': '7.2.0', 'pluggy': '1.0.0'}, 'Plugins': {'anyio': '3.6.2', 'asyncio': '0.21.0', 'cov': '4.0.0', 'html': '3.2.0', 'metadata': '2.0.4', 'mock': '3.10.0', 'JAVA_HOME': 'C:\\dev\\openjdk_18'}}
rootdir: D:\dev_workspaces\python_workspace\pytest-sample\tests, configfile: pytest.ini
plugins: anyio-3.6.2, asyncio-0.21.0, cov-4.0.0, html-3.2.0, metadata-2.0.4, mock-3.10.0
asyncio: mode-strict
collected 29 items / 28 deselected / 1 selected

tests\basic\test_calculator_module.py::test_sub_basic PASSED

===== 1 passed, 28 deselected in 0.26s =====
```

```
EXPLORER
PYTEST-SAMPLE
  .pytest_cache
  .vscode
  {} settings.json
  > htmlcov
  > src
  > tests
  > venv
  .coverage

pytest.ini
1 [pytest]
2 pythonpath = src
3 markers =
4   prod: production only tests (deselect with '-m "not prod"')
5   dev: dev only tests
6   qa: qa only tests
7
```

```
[pytest]
pythonpath = src
markers =
  prod: production only tests (deselect with '-m "not prod"')
  dev: dev only tests
  qa: qa only tests
```

9) 테스트 데이터가 자꾸 바뀌어서 관리하기 힘들어요

테스트 데이터 관리 Tips.

[테스트 데이터를 상수화해서 참조하기]

테스트 데이터는 종종 없어지거나 변경되기 쉽습니다.

방안1) 테스트에서 사용하는 테스트 데이터 값들은 별도로 한 파일에서 상수화해서 참조합니다

방안2) fixture 등의 별도 함수를 통해 테스트 수행 전 테스트 데이터를 확인하고, 미리 생성할 수 있습니다

[xxx_constants.py]

```
import os
import sys
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(
__file__))))

#### TestData ####
TEST_USER_ID = qa\_test001@lgresearch.ai
TEST_USER_PW = "automation123!"

#### API PATH ####
POST_USERREGISTER_APIPATH = "/account/register"
GET_PORTAL_IMAGELIST_APIPATH = "/web/portal"

GET_PORTAL_IMAGEDETAIL_APIPATH = "/web/portal/{PORTAL_IMAGE_ID}"
```

방안1) 테스트에서 사용하는 테스트 데이터 값들은 별도로 한 파일에서 상수화해서 참조합니다

[test_xxx_조회.py]

```
def data_fixture():
    # 미리 테스트 데이터를 생성

    yield

    # 테스트 데이터를 초기화

def test_xxx_01(data_fixture):
    # 테스트 데이터를 조회하고 확인
```

방안2) fixture 등의 별도 함수를 통해 테스트 수행 전 테스트 데이터를 확인하고, 미리 생성할 수 있습니다

그 외) pytest에서도 테스트 클래스를 사용하면 안 되나요?

pytest에서도 테스트 클래스 사용하기

[테스트 클래스]

pytest도 unittest처럼 테스트 클래스를 생성한 후 테스트 함수를 사용할 수도 있다

```
#-*- coding: utf-8 -*-
import os, sys, io
import pytest
from basic.calculator import calculator_machine
from common.exceptions import CalculateException

class TestCalculator:
    """ Test class for Calculator class """
    class_variable = "hello~ world~"

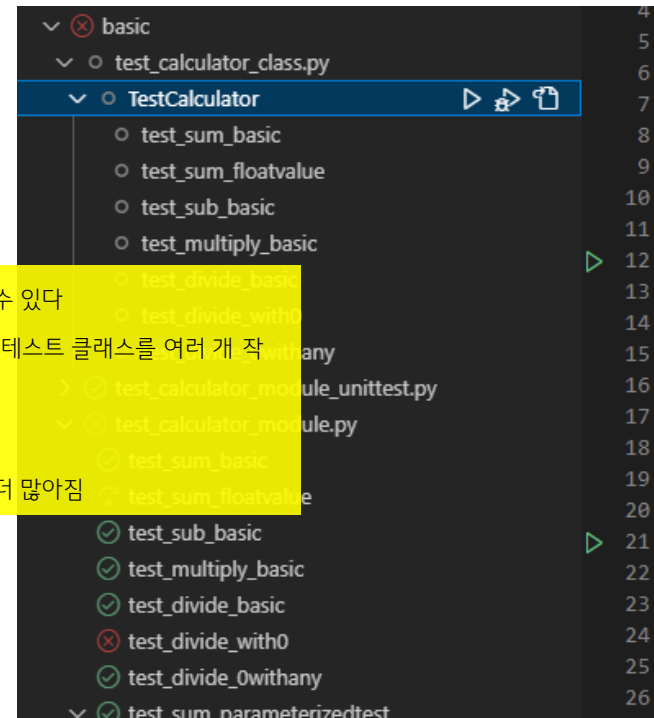
    def test_sum_basic(self):
        """ 테스트 목적 : 덧셈 기본확 """
        target = calculator_machine()
        a = 2
        b = 7
        result = target.sum(a,b)
        assert 9 == result
        assert "hello~ world~" == self.class_variable

    def test_sum_floatvalue(self):
        """ 테스트 목적 : 실수(소수점)에 대한 덧셈 시도 """
        target = calculator_machine()
        a = 2.2
        b = 7.1
        with pytest.raises(CalculateException) as float_error:
            target.sum(a,b)
        assert 'only int type is allowed' == str(float_error.value)
```

- pytest에서도 테스트 클래스를 선언한 후 테스트 함수를 작성할 수 있다

- 하나의 테스트 파일(의도)에서 여러 클래스를 테스트하려는 경우 테스트 클래스를 여러 개 작성하여 테스트 구성이 가능하다

- 대신 실제 테스트와 관계없는 템플릿성 코드 작성이 상대적으로 더 많아짐



10) 테스트를 얼마나 했는지 알고 싶어요

테스트 커버리지 - 라인, 브랜치

[pytest로 테스트 실행 후 pytest-cov로 테스트 커버리지 확인]

※ 테스트 커버리지 : 테스트 대상 중에 실제 얼마나 테스트를 수행했는지에 대한 지표. 테스트가 안 된 부분에 대해 테스트를 보완할 수 있다

※ 코드 테스트 커버리지 : 개발코드 상에서 테스트가 된/안 된 부분을 측정

(A) 라인 테스트 커버리지 : 개발 코드의 각 라인이 테스트되었는지. 예) 실행가능한 전체 라인 수 40lines 중 20lines이 실행 = 50%

(B) 브랜치 테스트 커버리지 : 개발 코드의 각 분기문(true/false등)에 대해 true, false 조건으로 각각 테스트 되었는지.

예) if문이 2개 있어서 각각 true/false 2개씩 총 4개의 분기 중 3개(true, true/false) 분기 실행 = 75%

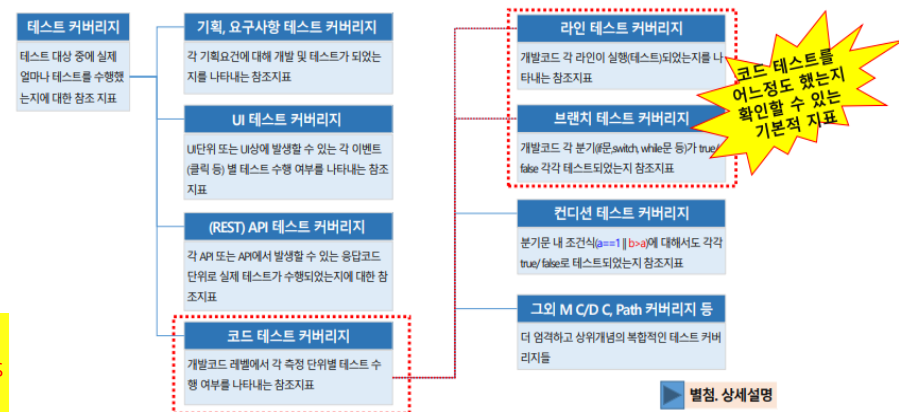
```
class calculator_machine:
    """
    """
    previous_value = 0

    def sum(self,a,b):
        """
        """
        if not isinstance(a,int) or type(b) is not int :
            raise CalculateException("only int type is allowed")
        if a<0 or b<0:
            raise CalculateException("only more than zero value can
input")
        return a+b+self.previous_value
```

sum 함수의 (A) 라인 수 : 6 lines

sum 함수의 (B) 브랜치 수 : 4 branches
if문 2개의 true/false 각각.

테스트를 잘 하도록 돕는 기법 - 테스트 커버리지



10) 테스트를 얼마나 했는지 알고 싶어요

[pytest로 테스트 실행 후 pytest-cov로 테스트 커버리지 확인]

1) vs code의 터미널에서 다음 pytest-cov 명령어와 함께 테스트를 실행한다

```
python -m pytest -v ./tests --junitxml="./TEST_RESULT.xml" --html="./test_result.html" --self-contained-html  
--cov-report html --cov-branch --cov=src
```

커버리지 리포트를 html 형태로 생성

브랜치 커버리지 측정

커버리지 측정 대상을 src 폴더 하위로 지정

```
----- coverage: platform win32, python 3.10.11-final-0 -----  
Coverage HTML written to dir htmlcov  
  
===== short test summary info =====  
FAILED tests/basic/test_calculator_class.py::TestCalculator::test_multiply_basic - common.exceptions.CalculateException: only int type is allowed  
FAILED tests/basic/test_calculator_class.py::TestCalculator::test_divide_with0 - AssertionError: assert 'only int type is allowed' == 'cannot divide with zero'  
FAILED tests/basic/test_calculator_class.py::TestCalculator::test_divide_0withany - TypeError: TestCalculator.test_divide_0withany() takes 0 positional arguments but 1 was given  
FAILED tests/basic/test_calculator_module.py::test_multiply_basic - common.exceptions.CalculateException: only int type is allowed  
FAILED tests/basic/test_calculator_module.py::test_divide_with0 - AssertionError: assert 'only int type is allowed' == 'cannot divide with zero'  
FAILED tests/mocking/test_alba.py::test_alba_poscalculate_basic - common.exceptions.ALBAException: 점장님~ 계산이 이상해요~  
FAILED tests/mocking/test_alba.py::test_alba_poscalculate_onlylatte - common.exceptions.ALBAException: 점장님~ 계산이 이상해요~  
FAILED tests/mocking/test_alba.py::test_alba_poscalculate_minus - common.exceptions.ALBAException: 점장님~ 계산이 이상해요~  
FAILED tests/mocking/test_alba_mocking.py::test_alba_poscalculate_basic - common.exceptions.ALBAException: 점장님~ 계산이 이상해요~  
===== 9 failed, 13 passed in 0.57s =====  
(venv) PS D:\dev_workspaces\python_workspace\pytest-sample >
```

이름	수정된 날짜	유형	크기
gitignore	2023-01-27 오후 3:52	텍스트 문서	1KB
coverage.html.js	2023-04-17 오후 4:09	JavaScript 파일	21KB
d_12f7c1a2832b7761__init__.py.html	2023-01-27 오후 3:52	Chrome HTML D...	5KB
d_12f7c1a2832b7761_fastapi_server.py...	2023-01-27 오후 3:52	Chrome HTML D...	5KB
d_400c6b07995ae146__init__.py.html	2023-04-03 오후 5:50	Chrome HTML D...	5KB
d_400c6b07995ae146_lam_logiccode.p...	2023-01-27 오후 3:52	Chrome HTML D...	5KB
d_400c6b07995ae146_lamalba.py.html	2023-04-17 오후 4:09	Chrome HTML D...	15KB
d_400c6b07995ae146_lamconstants.py...	2023-01-27 오후 3:52	Chrome HTML D...	5KB
d_400c6b07995ae146_menusupan.py.html	2023-04-03 오후 5:50	Chrome HTML D...	6KB
d_5895cd8a5bb85824__init__.py.html	2023-01-27 오후 3:55	Chrome HTML D...	5KB
d_5895cd8a5bb85824_exceptions.py.ht...	2023-04-03 오후 5:50	Chrome HTML D...	12KB
d_9732a27fcd4ccd2f__init__.py.html	2023-04-03 오후 5:50	Chrome HTML D...	5KB
d_9732a27fcd4ccd2f_calculator.py.html	2023-04-17 오후 4:09	Chrome HTML D...	23KB
d_bd9ac6f3af0f792e__init__.py.html	2023-01-27 오후 3:55	Chrome HTML D...	5KB
d_bd9ac6f3af0f792e_calculat...	2023-01-27 오후 3:55	Chrome HTML D...	12KB
d_d52bce7b5dc3866b__init...	2023-04-17 오후 4:09	Chrome HTML D...	5KB
d_d52bce7b5dc3866b_lam...	2023-04-17 오후 4:09	Chrome HTML D...	5KB
favicon_32.png	2023-04-17 오후 4:09	웹용 PNG 파일	2KB
index.html	2023-04-17 오후 4:09	Chrome HTML D...	8KB
keybd_closed.png	2023-04-17 오후 4:09	알짜 PNG 파일	9KB
keybd_open.png	2023-04-17 오후 4:09	알짜 PNG 파일	9KB
status.json	2023-04-17 오후 4:09	JSON 원본 파일	4KB
style.css	2023-04-17 오후 4:09	CSS 스타일시트 ...	13KB

/htmlcov/index.html

Coverage report: 76%

coverage.py v7.0.5, created at 2023-06-20 17:43 +0900

전체 리포트

Module	statements	missing	excluded	branches	partial	coverage
src\basic__init__.py	0	0	0	0	0	100%
src\basic\calculator.py	30	6	0	16	6	74%
src\coffee_restservice__init__.py	0	0	0	0	0	100%
src\coffee_restservice\fastapi_server.py	47	11	0	6	1	74%
src\coffee_service__init__.py	0	0	0	0	0	100%
src\coffee_service\iamalba.py	28	6	0	10	0	100%
src\coffee_service\menusupan.py	2	0	0	0	0	100%
src\common__init__.py	0	0	0	0	0	100%
src\common\exceptions.py	17	2	0	0	0	88%
Total	124	25	0	32	7	76%

(A)Line Coverage : 76% = 99/124lines

10) 테스트를 얼마나 했는지 알고 싶어요

- 2) 결과 디렉토리에 생성된 html 커버리지 리포트에서 상세 코드별로 테스트가 안 된 부분을 확인한다
- 3) 테스트가 안 된 부분에 대해 추가 테스트를 작성할 수 있다

[FAQ]

```
Coverage for src\basic\calculator.py: 74%
30 statements 24 run 6 missing 0 excluded 6 partial
« prev ^ index » next coverage.py v7.0.5, created at 2023-06-20 17:43 +0900

1  #-*- coding: utf-8 -*-
2  import os,sys
3  from common.exceptions import CalculateException
4  # sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
5
6  class calculator_machine:
7      """
8      """
9      previous_value = 0
10
11     def sum(self,a,b):
12         """
13         """
14         if not isinstance(a,int) or type(b) is not int :
15             raise CalculateException("only int type is allowed")
16         if a<0 or b<0:
17             raise CalculateException("only more than zero value can input")
18         return a+b+self.previous_value
19
20     def subtract(self,a,b):
21         """making this more complex to view sonarqube issues reporting """
22         if not isinstance(a,int) or type(b) is not int :
23             raise CalculateException("only int type is allowed")
24         else:
25             if a<0:
26                 raise CalculateException("only more than zero value can input")
27             else:
28                 if b<0:
29                     raise CalculateException("only more than zero value can input")
30             return a-b+self.previous_value
31
32     def multiply(self,a,b):
33         """
34         """
35         # if a is not int or type(b) is not int :
36         if not isinstance(a, int) or type(b) is not int :
37             raise CalculateException("only int type is allowed")
38         return a*b+self.previous_value
39
40     def divide(self,a,b):
41         """making this more complex to view sonarqube issue reporting """
42         if not isinstance(a,int) or type(b) is int :
43             if b==0:
44                 raise CalculateException("cannot divide with zero")
45             else:
46                 return a/b+self.previous_value
```

소스 코드 상에

(a) 실행된 코드 : 녹색

(b) 실행 안 된 코드 : 빨간 색

(c) 부분적으로 실행된 코드 : 노란 색

표시

테스트는 실패했는데 왜 커버리지가 나오나요?

테스트 커버리지는 단순히 코드가 실행됐다 안 됐다를 표시해 줄 뿐 테스트의 성공/실패와는 관계가 없습니다

테스트 커버리지는 몇 %까지 맞춰야 하나요?

테스트 커버리지는 단순 참고지표이고, 각 코드, 도메인, 기술별로 중요도가 달라서 절대적인 숫자는 없습니다. 대신 일반적으로는 라인 커버리지 70% 정도를 적정한 수준으로 얘기하는 경우가 많습니다

별도 html리포트가 아닌 개발 IDE에서 코드에 바로 커버리지 표시는 안 되나요?

(다음 장) Coverage Gutters Extension을 깔아보겠습니다!!

10) 테스트를 얼마나 했는지 알고 싶어요

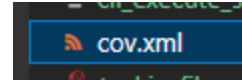
VS Code에서 바로 커버된 라인 확인하기

1) Extension에서 'Coverage Gutters'를 검색해 설치합니다

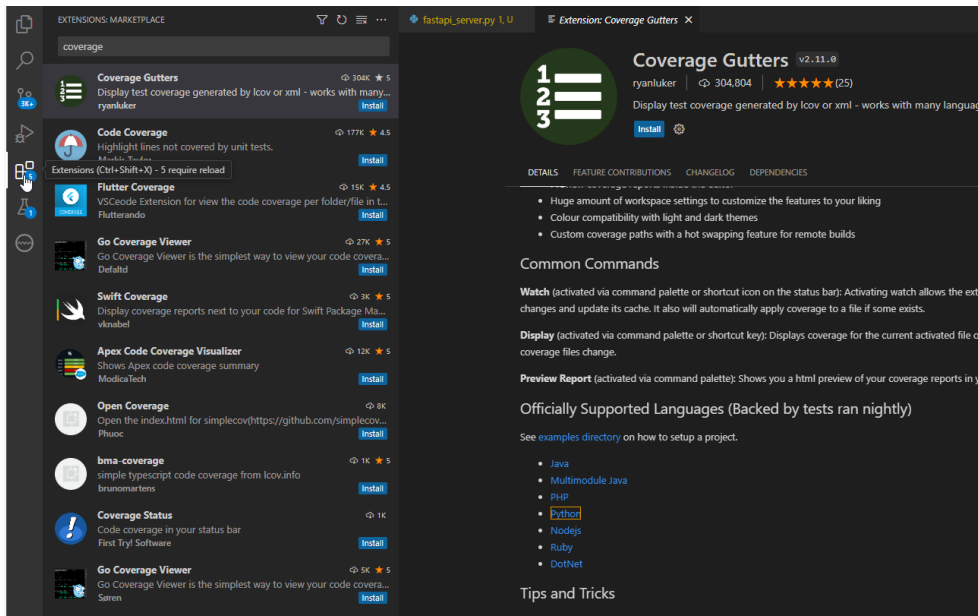
2) Pytest를 다음 명령어로 실행합니다 (cov.xml 파일 생성)

```
$ python -m pytest ./tests --cov-report xml:cov.xml --cov=src
```

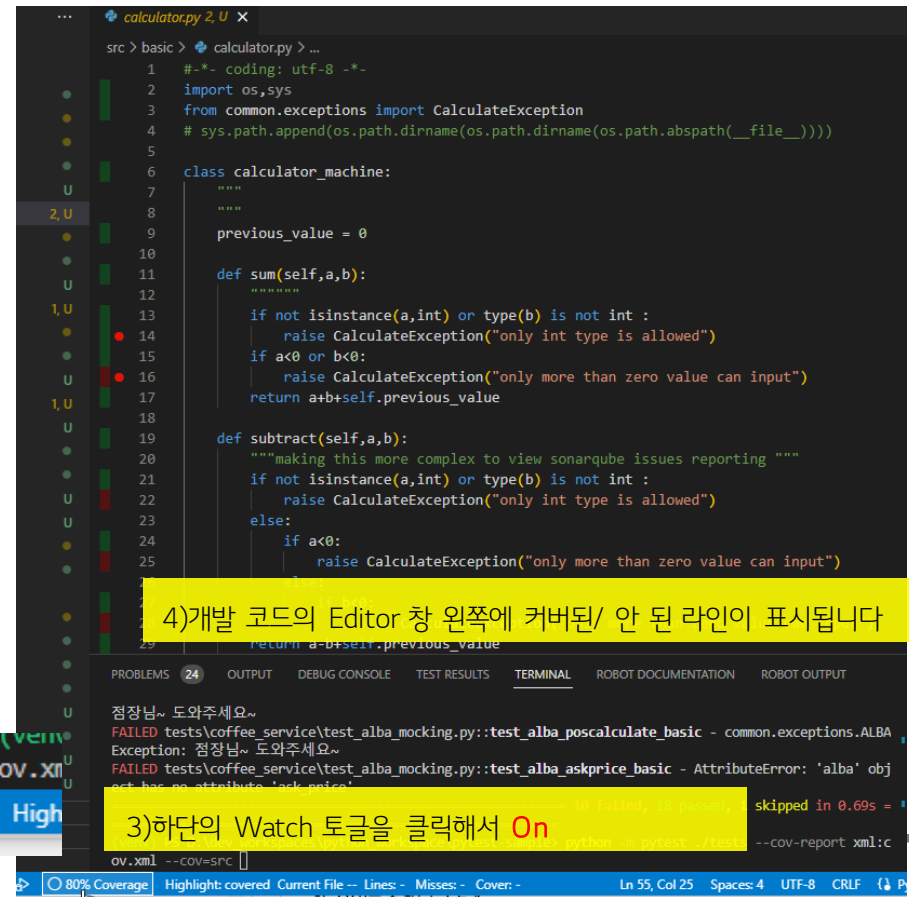
3) VS Code 하단의 Watch를 체크하고 개발코드를 VS Code에서 선택합니다



2)Pytest 실행 시 결과를 xml(cov.xml) 형태로 출력



1)Extension에서 'Coverage Gutters'를 검색해 설치



4)개발 코드의 Editor 창 왼쪽에 커버된/ 안 된 라인이 표시됩니다

3)하단의 Watch 토글을 클릭해서 On

1. 개요

2. Pytest 기본 사용법

3. Mocking을 통한 단위 테스트

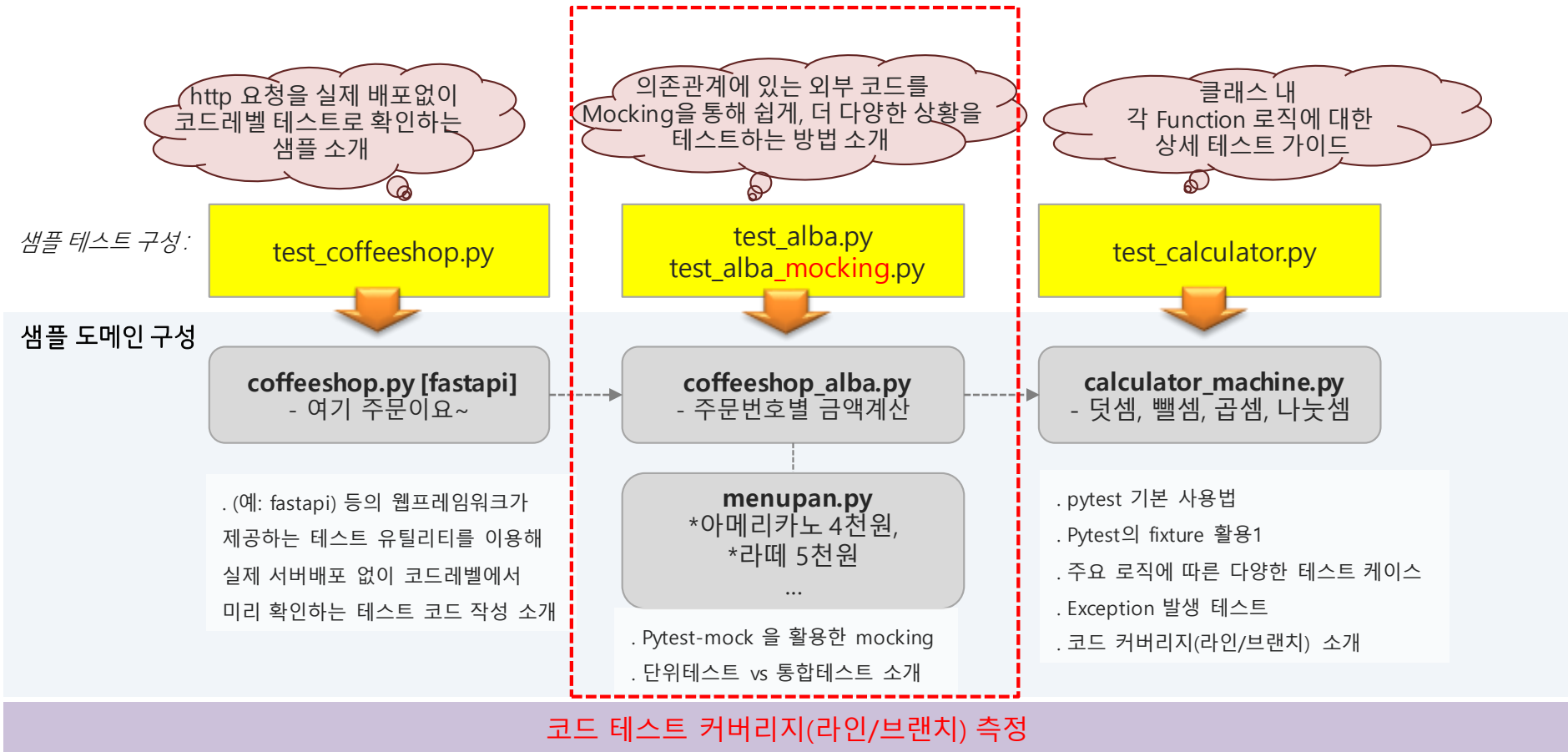
4. 코드레벨 통합 테스트(HTTP API 테스트)

5. 정리

가이드 구성 - 목적/상황별

[샘플/가이드 구성안 - 초안]

- 계산기(AI 모델 모듈)를 이용해서 알바생(플랫폼 로직코드)이 커피 값을 계산하는 코드를 http(XXX서버)로 제공하는 서비스에 대해
- 각각 (a)알고리즘 코드 자체에 대한 테스트, (b,b')참조하는 모듈을 Mocking하며 하는 테스트, (c)실제 서버 배포 전에 코드레벨에서 http 요청을 확인하는 테스트 샘플을 제공



Mock을 이용한 단위 상세 테스트

Mock이란?

실제 객체 대신에 가짜 객체를 만들어 사용하는 방법

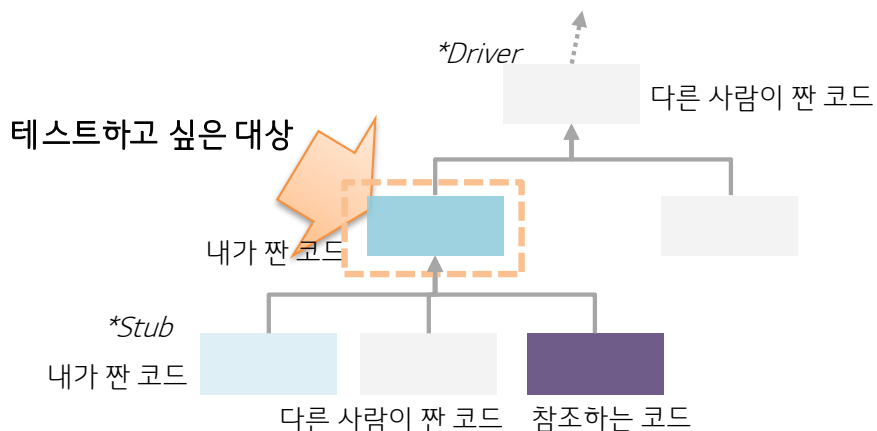
Mock 객체는 언제 필요한가?

- 테스트 작성을 위한 환경 구축이 어려운 경우
- 테스트가 특정 경우나 순간에 의존적인 경우
- 테스트 시간이 오래 걸리는 경우

테스트 하려는 코드가 의존하고 있는 객체를 가짜로 만들어 의존성 제거하고 객체의 동작을 통제할 수 있다.

의존성이 있는 코드를 테스트하다가 테스트를 실패할 경우 어떤 코드가 문제인지 모르게 된다. 의존성 객체를 모킹함으로써 테스트 중인 코드에만 집중하여 테스트할 수 있다.

단위 테스트를 할 때는 테스트 하려는 객체에만 집중하고, 통합 테스트를 통해서 전체적인 테스트를 할 수 있다



※ Mock 테스트가 필요한 실제 상황 예.

1) 재현하기 어려운 상황들에 대한 Mock 처리

- AI 모델에서 GPU 메모리 오류 응답에 대해 에러 응답 처리 – 동시에 10명 사용자가 10여분 이상 반복 호출

2) 전체 통합테스트에서는 찾기 어려운 오류나 오류 원인

HTTP API에서 전달한 여러 파라미터가 AI 추론 시 적용이 안 되는 것 같은 현상 -> 중간에 코드 오류로 첫번째 파라미터 값을 모두 사용함

2') AI 결과 개수가 기대한 것보다 적게 나왔을 때 어떤 과정에서 적게 나오게 됐는지 디버깅

- AI 결과 자체가 중복된 내용이 나와서 생략된 것인지 vs 결과를 정제하는 과정에서 의미있는 문장이 안 나와서 (예: ".") 개수가 줄어든 것인지 확인

<= 최종 결과에 대해 중간, 중간 결과를 확인 가능하여 실제 오류인지, 어떤 부분의 오류인지 파악이 쉬워 짐(단위 테스트)

Pytest-mock 사용하기

Pytest-mock

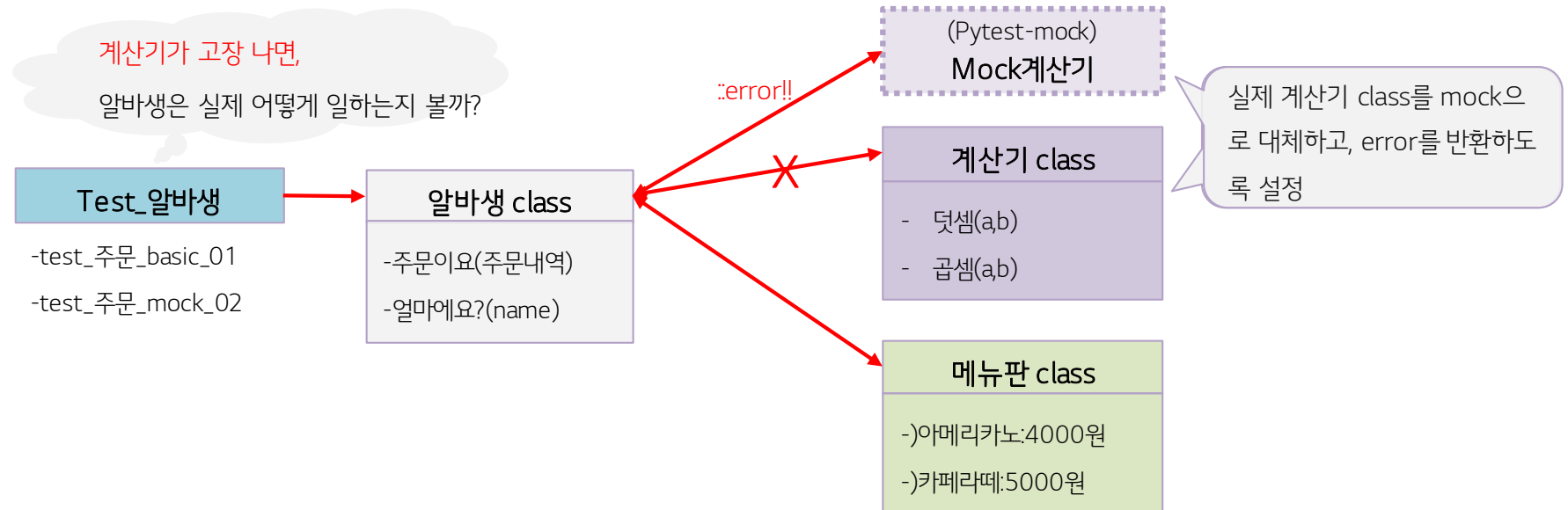
Pytest-mock

이 pytest 플러그인은 mocker fixture를 제공하여 mock 기능을 지원합니다

설치

\$ pip install pytest-mock

<https://pytest-mock.readthedocs.io/en/latest/>



Pytest-mock 사용하기

Pytest-mock

- 1) Pytest-mock 설치 후 “mock”를 fixture를 주입하면, Mocker 기능을 사용할 수 있습니다
- 2) mocker.patch로 mock 동작을 미리 정의하고
 - (a) 특정 값 반환 : mocker.patch(“패키지.모듈.클래스.함수이름”, return_value=~~)
- 3) 테스트를 실행하면, 런타임에 참조하는 함수가 mocker.patch로 미리 정의한 값을 반환합니다

1) Pytest-mock 설치 후 “mock”를 fixture를 주입하면, Mocker 기능을 사용할 수 있습니다

```
def test_alba_poscalculate_basic(mock):
    """ 테스트 목적: 계산기 핑계를 대는 알바생에게 바로 계산해보라고 시키기 """
    target = alba()
    purchased_drinks={
        "AMERICANO" : 3,
        "CAFFELATTE" : 2
    }
    # mocker.spy(calculator_machine, "multiply")
    mocker.patch("basic.calculator.calculator_machine.multiply", return_value=10000)    #4000, 5000
    mocker.patch("basic.calculator.calculator_machine.sum", return_value=22000)    #4000, 5000
    result = target.pos_calculate(2,purchased_drinks)
    assert 22000 == result
```

2) mocker.patch로 mock 동작을 미리 정의하고

(a) 특정 값 반환 : mocker.patch(“패키지.모듈.클래스.함수이름”, return_value=~~)

3) 테스트를 실행하면, 런타임에 참조하는 함수가 mocker.patch로 미리 정의한 값을 반환합니다

Pytest-mock 사용하기

Pytest-mock

(b) 특정 **Exception** 발생 : `mock.patch("패키지.모듈.클래스.함수이름", side_effect=XXXException("계산기 오류입니다"))`

```
def test_alba_poscalculate_calculatorerror(mock):
    """ 테스트 목적: 계산기가 임의로 CalculateException 발생할 때 알바생의 동작을 확인 """
    target = alba()
    purchased_drinks={
        "AMERICANO" : 3,
        "CAFELATTE" : 2
    }
    mock.patch("basic.calculator.calculator_machine.sum", side_effect=CalculateException("계산기 오류입니다"))
    with pytest.raises(ALBAException) as expected_saying:
        target.pos_calculate(2,purchased_drinks)
    assert "점장님~ 계산이 이상해요~" == str(expected_saying.value)
```

(b) 특정 **Exception** 발생 : `mock.patch("패키지.모듈.클래스.함수이름", side_effect=XXXException("계산기 오류입니다"))`

(c) 상수 값 치환 : `mock.patch.object(클래스, "치환하려는 상수 이름", 바꾸려는 값)`

```
def test_alba_askprice_basic(mock):
    """ 테스트 목적: 가격 물어보기. 이미 정의된 상수 값을 임의로 조작한 후 확인하는 예제 """
    to_ask_beverage = "CAFELATTE"
    target = alba()
    from coffee_service import menupan
    mock.patch.object(menupan, "PRICE_CAFELATTE", 10000)
    answer_price = target.ask_price(to_ask_beverage)
    assert '카페라떼는 10000 원입니다' == answer_price
```

(c) 상수값 치환 : `mock.patch.object(클래스, "치환하려는 상수 이름", 바꾸려는 값)`

※ 더 많은 활용 방법

<https://pytest-mock.readthedocs.io/en/latest/usage.html>

- mock.patch : 함수를 mocking
- mock.patch.object : 클래스를 mocking
- mock.patch.multiple : 한 번의 호출로 여러 패치를 수행
- mock.patch.dict : dictionary를 패치
- mock.stopall : mock 패치를 정지하고 원래 코드를 실행
- mock.stop : patcher에서 정의한 대상의 mockin을 start 또는 stop
- Mocker.resetall(): 모든 mock 오브젝트에 대해 reset_mock() 호출

또한 pyest-mock의 mocker로 부터 다음의 unittest mock 모듈들도 접근 가능하다

- Mock
 - MagicMock
 - PropertyMock
 - ANY
 - DEFAULT
 - Call
 - Sentinel
 - mock_open
 - seal
-

※ 참고) pytorch에서의 코드 테스트?

[PyTorch's existing testing tools]

<https://github.com/pytorch/pytorch/wiki/Running-and-writing-tests>

<https://labs.quansight.org/blog/2021/06/pytest-pytorch>

PyTorch's test framework lets you instantiate test templates for different operators, datatypes (dtypes), and devices to improve test coverage. It is recommended that all tests be written as templates, whether it's necessary or not, to make it easier for the test framework to inspect the test's properties.

- Common test utilities
- PyTorch's test generation functionality
- OpInfos

[Pytest-pytorch]

몇 가지 pytest에서 유효하지 않은 네이밍을 갖는 pytorch 테스트 기능을 지원하는 라이브러리

<https://pypi.org/project/pytest-pytorch/>

```
%%run_pytest[clean] {MODULE}

import torch

from torch.testing import _dispatch_dtypes
from torch.testing._internal.common_device_type import (
    instantiate_device_type_tests,
    ops,
)
from torch.testing._internal.common_methods_invocations import OpInfo
from torch.testing._internal.common_utils import TestCase

op_db = [
    OpInfo("add", dtypesIfCPU=_dispatch_dtypes([torch.int32])),
    OpInfo("sub", dtypesIfCPU=_dispatch_dtypes([torch.float32])),
]

class TestFoo(TestCase):
    @ops(op_db)
    def test_bar(self, device, dtype, op):
        pass

instantiate_device_type_tests(TestFoo, globals(), only_for="cpu")

tmp119_vdl.py::TestFooCPU::test_bar_add_cpu_int32
tmp119_vdl.py::TestFooCPU::test_bar_sub_cpu_float32

2 tests collected in 0.04s
```

※ 참고) pytorch에서의 코드 테스트?

※ 단위테스트 : 내가 테스트하려는 대상만을 독립시키고 상세하게 테스트

※ 통합테스트 : 실제 동작과 유사하게 여러 단위들을 통합하여 기대한 동작 확인

[Pytorch 단위테스트, 무엇을 테스트해야 할까?]

<https://krokotsch.eu/posts/deep-learning-unit-tests/>

Dataset

- The shape of your data
- The scale of your data
- The augmentation of your data
- The loading of your data

Model

- The output shape of your mode
- The moving of your model
- The sample independence of your model
- The parameter updates of your model
- Improving reusability

Trainer

- The loss of your trainer
 - The logging of your trainer
 - The fitting of your trainer
-

1. 개요

2. Pytest 기본 사용법

3. Mocking을 통한 단위 테스트

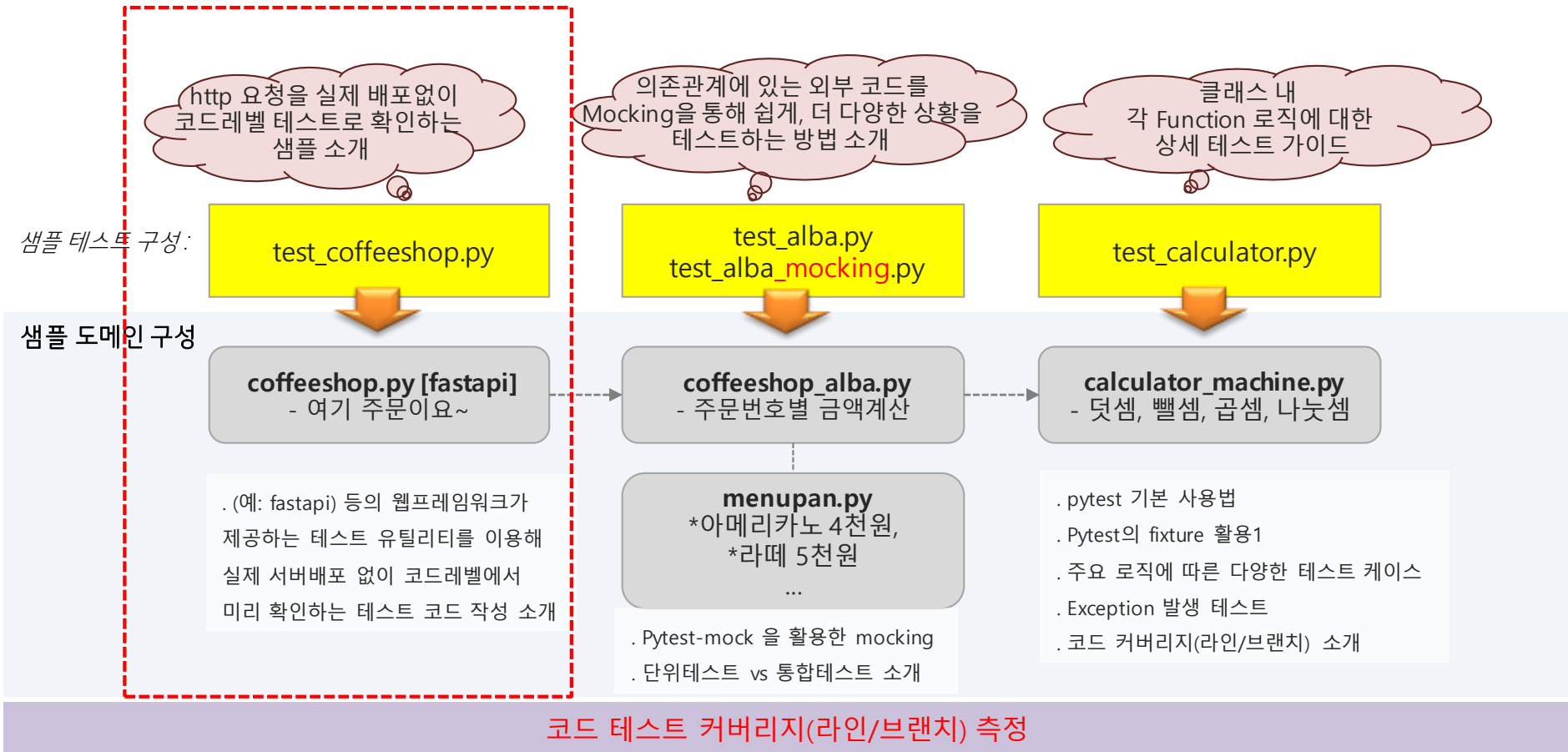
4. 코드레벨 통합 테스트(HTTP API 테스트)

5. 정리

가이드 구성 - 목적/상황별

[샘플/가이드 구성안 - 초안]

- 계산기(AI 모델 모듈)를 이용해서 알바생(플랫폼 로직코드)이 커피 값을 계산하는 코드를 http(XXX서버)로 제공하는 서비스에 대해
- 각각 (a)알고리즘 코드 자체에 대한 테스트, (b,b')참조하는 모듈을 Mocking하며 하는 테스트, (c)실제 서버 배포 전에 코드레벨에서 http 요청을 확인하는 테스트 샘플을 제공



코드레벨 통합 테스트(HTTP API 테스트)

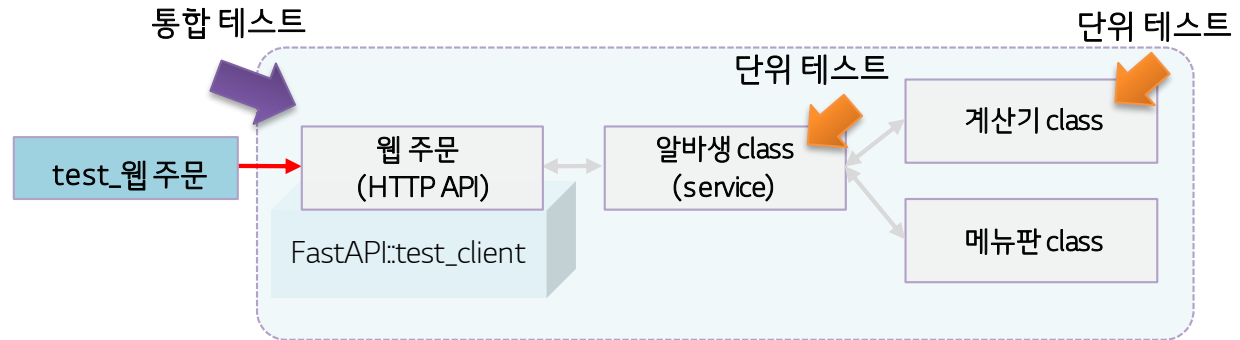
테스트 영역 - 코드 레벨 통합 테스트

FastAPI 등 웹 프레임워크에서 제공하는 test_client를 이용하여

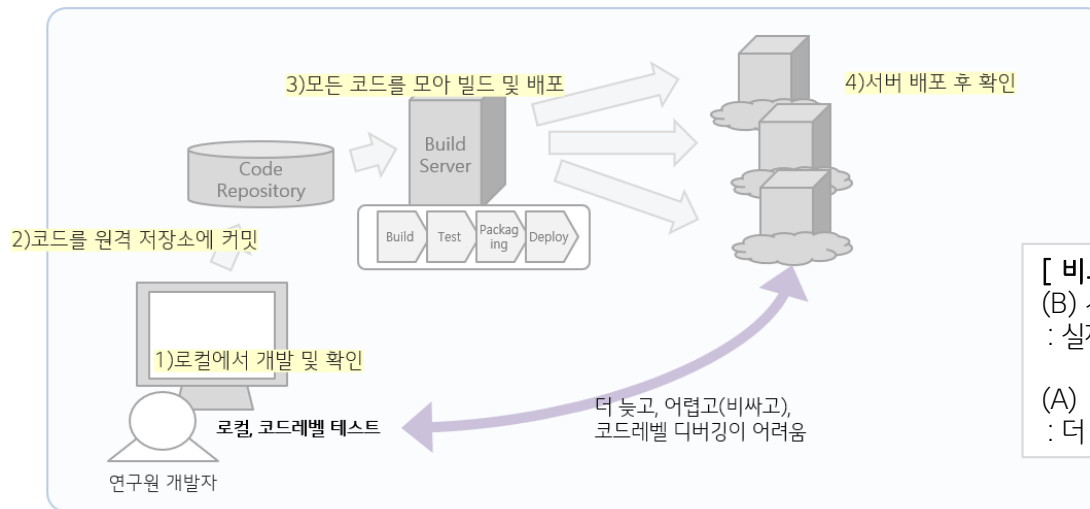
직접 서버를 띄우지 않고도 HTTP API 테스트를 수행

```
client = fastapi.testclient.TestClient(app)

client.get("/price?item=Americano")
assert 200 == response.status_code
assert "4000원입니다" == response.json()['result']
```



테스트 시점 - 코드 통합 및 서버 배포 전 테스트



[비교]

(B) 서버 배포 후 API 테스트 : 사용자, API 스펙 관점 테스트
: 실제 서비스되는 환경과 제일 가까운 테스트

(A) 코드레벨 API 테스트 : 개발자/화이트박스 코드 관점 테스트
: 더 빠른 시기에, 쉽게, 테스트 디버깅 등의 장점을 유지하며 전체 통합 테스트 수행

HTTP API 테스트

HTTP API 테스트

FASTAPI가 제공하는 test_client를 통해 HTTP API 테스트

FastAPI에서 제공하는 TestClient 객체를 사용하면 위에서 작성한 API 코드를 client 변수를 통해 요청하고 응답 코드 및 json 결과값을 테스트할 수 있습니다

- 서버에 배포하지 않고도 제공하는 HTTP API에 대한 테스트를 할 수 있습니다
- 디버그, 테스트 커버리지 등을 그대로 활용할 수 있습니다
- 코드레벨에서는 최상위 통합 테스트입니다

설치

FastAPI의 test_client는 기본 패키지로 포함되어 있어 별도 설치가 필요 없습니다

```
$ pip install fastapi
```

```
$ pip install uvicorn
```

```
from fastapi.testclient import TestClient
from coffee_restservice.fastapi_server import app

client = TestClient(app)
```

HTTP API 테스트

HTTP API 테스트

FastAPI의 TestClient를 활용하면 직접 서버 배포 없이도 HTTP 요청이 가능합니다

- 1) fastapi.testclient의 TestClient 클래스를 import 한다
- 2) 작성한 FastAPI app을 TestClient의 인자로 전달한다
- 3) TestClient는 기존 pytest 기능을 그대로 지원한다

TestClient에 get, post 등 작성한 HTTP API 를 요청하고 응답을 확인합니다

1) fastapi.testclient의 TestClient 클래스를 import

```
from fastapi.testclient import TestClient
from coffee_restservice.fastapi_server import app
```

```
client = TestClient(app)
```

2) 작성한 FastAPI app을 TestClient의 인자로 전달

```
def test_fastapiserver_askprice_americano():
    """ 테스트 목적 : 아메리카노, 카페라떼 가격 문의 """
    to_ask_item = "AMERICANO"
    response = client.get("/price", params = {"item":to_ask_item})
    assert 200 == response.status_code, response.text
    response_body = response.json()
    assert 'result' in response_body
    assert '아메리카노는 4000 원입니다' == response_body['result']
```

3) TestClient에 get, post 등 작성한 HTTP API 를 요청하고 응답을 확인

HTTP API 테스트

※ (비교) 실제 서버 배포 후 호출 확인

FastAPI의 TestClient를 활용하면 직접 서버 배포 없이도 HTTP 요청이 가능합니다

(실제 서버 배포)

1) 직접 FastAPI 서버를 로컬에 띄우고

```
$ uvicorn src.coffee_restservice.fastapi_server:app --reload
```

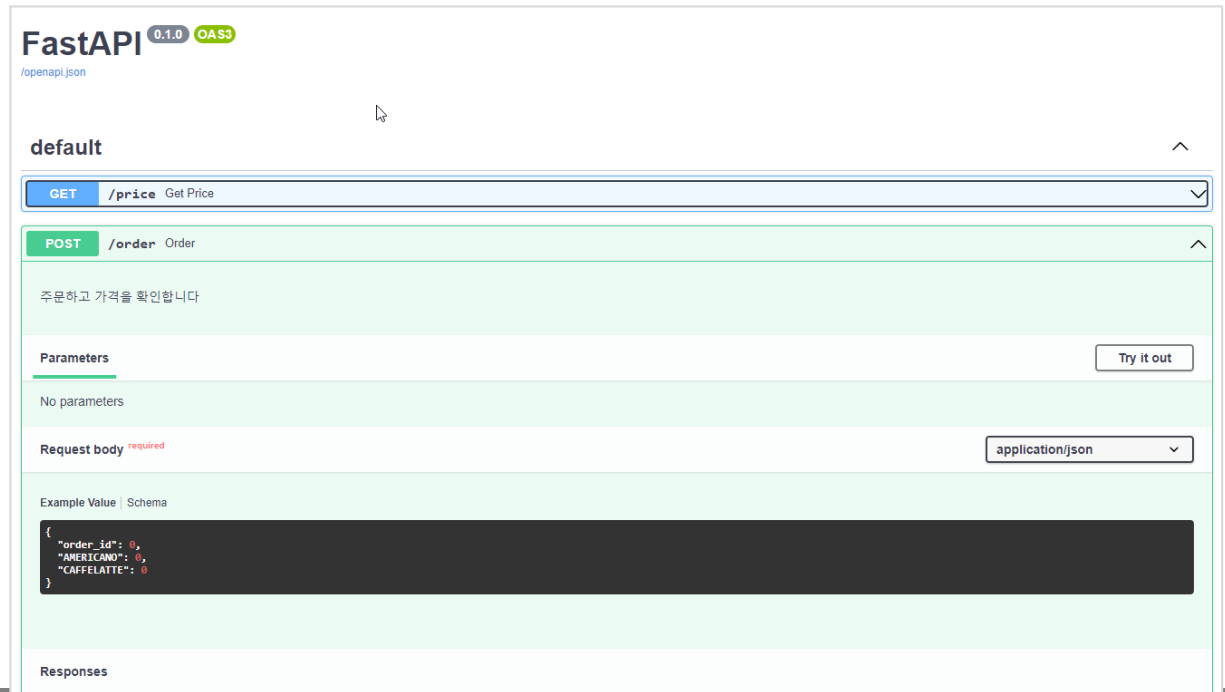
2) swagger를 확인한 모습

<http://127.0.0.1:8000/docs>

[FastAPI 어플리케이션 코드]

```
fastapi_server.py 1,0 X
src > coffee_restservice > fastapi_server.py > get_price
17
18 app = FastAPI()
19 common_error_msg = "오류가 발생했습니다. 잠시만 기다려 주세요."
20
21 @app.get("/price", description="아이템의 가격을 문의합니다", status_code=200)
22 async def get_price(item:str="AMERICANO,CAFFELATTE"):
23     """ 쿼리파라미터로 item="AMERICANO" 등으로 가격을 묻습니다
24     """
25     albasang = alba()
26     if item not in ['AMERICANO', 'CAFFELATTE']:
27         raise HTTPException(status_code=400, detail="판매하지 않는 음료입니다")
28     try:
29         answer = albasang.ask_price(item)
30         return _make_200_response(answer)
31     except ALBAException as alba_saying:
32         raise HTTPException(status_code=500, detail=str(alba_saying.value))
33     except Exception as detail_chk:
34         raise HTTPException(status_code=500, detail=common_error_msg)
35
36
37 @app.post("/order", description="주문하고 가격을 확인합니다", status_code=201) # GET
38 async def order(order: Order): # root() 함수를 실행하고
39     """ 주문하고 가격 확인
40     """
41     albasang = alba()
42     try:
43         order_detail = {
44             "AMERICANO": order.AMERICANO,
45             "CAFFELATTE": order.CAFFELATTE
46         }
47         return _make_response(fastapi.status.HTTP_201_CREATED, "전체 가격은 {}원입니
48     except ALBAException as alba_saying:
49         raise HTTPException(status_code=500, detail=str(alba_saying.value))
50     except Exception as detail_chk:
51         raise HTTPException(status_code=500, detail=common_error_msg)
52
53
```

[FastAPI서버를 띄운 후 swagger 확인]



HTTP API 테스트

HTTP API 테스트

[Get 방식 샘플 테스트] : (GET) /price?item=AMERICANO

```
from fastapi.testclient import TestClient
from coffee_restservice.fastapi_server import app

client = TestClient(app)

def test_fastapiserver_askprice_americano():
    """ 테스트 목적 : 아메리카노, 카페라떼 가격 문의 """
    to_ask_item = "AMERICANO"
    response = client.get("/price", params = {"item":to_ask_item})
    assert 200 == response.status_code, response.text
    response_body = response.json()
    assert 'result' in response_body
    assert '아메리카노는 4000 원입니다' == response_body['result']
```

[Post 방식 샘플 테스트] : (POST) /order

```
def test_fastapiserver_order_basic():
    data = {
        "order_id" : 1,
        "AMERICANO" : 1,
        "CAFELATTE" : 1
    }
    response = client.post("/order", json = data)
    assert 201 == response.status_code, response.text
    response_body = response.json()
    assert 'result' in response_body
    assert '전체 가격은 9000원입니다' == response_body['result']
```

※ Delete는 Get 방식과, PUT은 POST 방식과 유사

※ HTTP API에 대해서도 개발 IDE내에서 디버깅이 가능하고,
코드 테스트 커버리지 확인도 가능함

그 외 : pytest-asyncio

그 외 : 비동기 테스트

pytest-asyncio

HTTP 응답이 바로 오지 않고 지연이 발생하는 비동기 테스트의 경우, pytest-asyncio를 활용하여 테스트를 작성합니다

이 경우 앞에서 사용한 FastAPI의 TestClient에서는 지원하지 않기 때문에 아래와 같이 HTTPX 패키지의 AsyncClient를 사용합니다

설치

```
$ pip install pytest-asyncio
```

```
$ pip install httpx
```

테스트 방법

비동기 테스트를 위해서는 함수명 위에

`@pytest.mark.asyncio`를 명시해주면 pytest가 해당 테스트를 비동기로 처리해주게 됩니다.

그런 다음 AsyncClient와 await를 이용하여 비동기 요청을 보내고 응답을 받을 수 있습니다.

(참고) <https://sehoi.github.io/etc/fastapi-pytest/>

```
import json
import pytest

from httpx import AsyncClient

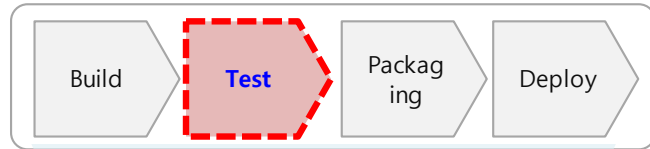
@pytest.mark.asyncio
async def test_root():
    async with AsyncClient(base_url="http://127.0.0.1:8000") as ac:
        response = await ac.get("/")
        assert response.status_code == 200
        assert response.json() == {"msg": "Hello World"}
```

-
1. 개요
 2. Pytest 기본 사용법
 3. Mocking을 통한 단위 테스트
 4. 코드레벨 통합 테스트(HTTP API 테스트)
 5. 정리

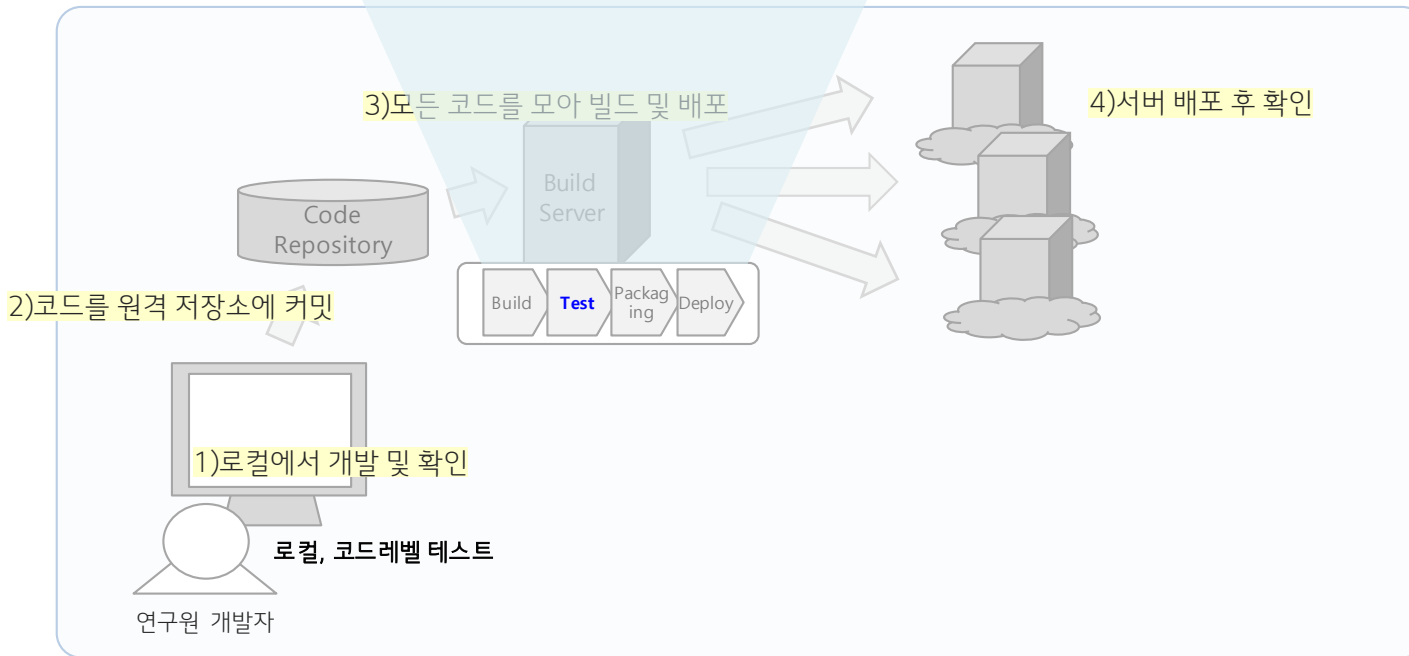
빌드 파이프라인 상에서의 pytest

빌드/배포 파이프라인에서 자동화된 테스트

- 작성한 테스트는 빌드/배포 파이프라인상에서 재사용되어 지속적인 테스트와 배포의 한 구성요소가 된다



pytest_jenkins_slack_demo.mp4



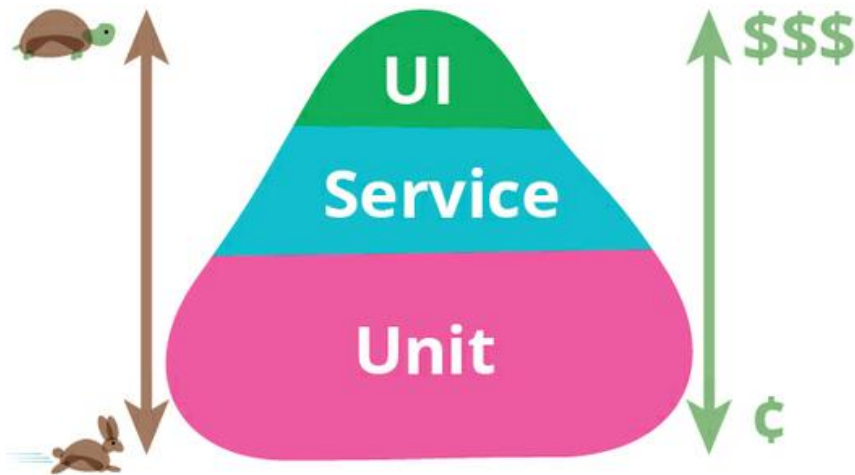
단위, 화이트박스 테스트, 코드 테스트

화이트박스 / 코드레벨 테스트의 장점

- 개발 코드 작성과 동시에 숨겨진 오류를 쉽게 찾을 수 있습니다 (디버깅)
- 작성한 테스트 코드를 통해 테스트 케이스 재사용성 및 안정성 향상
- 자동화하기 가장 쉬운 테스트
- 코드 리팩토링, 최적화 촉진
- 더 상세한 테스트가 가능
- 코드 테스트 커버리지 확인
- 더 빠른 테스트(time-saving)

화이트박스 / 코드레벨 테스트 주의할 점

- 과도한 mocking은 지양
- 테스트 커버리지는 참고 지표이지 목표 지표가 아님 (100%달성 X)
- 단위와 통합테스트가 각 목적에 맞게 어우러져야 함



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

“(참고) 테스트 자동화 피라미드”

- By Martin Fowler 등 여러 사람들&오랜 기간 얘기되는 개념.

(a) UI 테스트, (b) API/Service 테스트, (c) Unit 테스트

=> 테스트 자동화 피라미드의 위로 올라갈 수록 테스트에 들어가는 비용도 비싸고, 작성하기 어렵고, 수행 시간도 더 오래 걸린다

=> 따라서 각 테스트 레벨의 영역(넓이)만큼의 테스트가 존재해야 한다

교육 회고

이 교육을 통해서,

- Python의 화이트박스 테스트 도구인 pytest의 기본 사용법을 배우고, 내 개발 코드/프로젝트에 적용할 수 있습니다

Yes!!

- 작성한 테스트 코드를 이용하여 개발 IDE내에서 손쉽게 디버깅을 할 수 있습니다

Yes!!

- 테스트 커버리지, 단위테스트/통합테스트의 개념과 이점을 이해합니다

Yes!!

- Mock 테스트가 필요한 경우를 이해하고, 내 개발 코드에 적용할 수 있습니다

Yes!!

- HTTP API에 대해서도 손쉽게 테스트 코드를 작성하고 확인할 수 있습니다

Yes!!

- The end -

감사합니다.
