Python 기반의 단위 테스트

㈜ 시네틱스

1 Python Black

Black

❑ 도구 개요

- 코드 스타일을 통일시켜주는 코드 포맷팅 도구(Code Formatter)

❑ 필요성

- 협업을 위한 약속
- 읽기 좋은 코드가 좋은 코드
- 스타일 기준에 대한 고민 줄이기

❑ 사용 방법

- Command-Line에서 문제 파일 찾고 Black이 수정하기
- (권고) IDE에서 파일 저장할 때마다 Black이 수정하기

Black 설치

❑ PIP를 이용한 설치

- pip install black

```
PS C:\Users\devops\DevOps_Test> pip install black
Defaulting to user installation because normal site-packages is not
 writeable
Collecting black
  Downloading black-22.1.0-cp310-cp310-win_amd64.whl (1.1 MB)
     |                                | 1.1 MB 1.3 MB/s
Collecting mypy-extensions>=0.4.3
  Downloading mypy_extensions-0.4.3-py2.py3-none-any.whl (4.5 kB)
```

❑ 설치 확인

- black

```
PS C:\Users\djhan\Prj> black
Usage: black [OPTIONS] SRC ...

One of 'SRC' or 'code' is required.
```

Command-Line으로 사용하기

❑ 포맷팅 수정할 파일이 있는지 확인하기

- black --check [파일명] or [폴더명]
- 예) black --check .
  - 현재 폴더에 포맷팅 수정할 파일이 있는지 확인

```
PS C:\Users\djhan\Prj> black --check .
would reformat apitemp.py
would reformat calc.py
would reformat black_example.py
would reformat testSum.py
would reformat testSum2.py
```

❑ 포맷팅 수정하기

- black [파일명] or [폴더명]
- 예) black example.py

```
PS C:\Users\djhan\Prj> black testSum2.py
reformatted testSum2.py

All done! ✨ 🍰 ✨
1 file reformatted.
```

```
if very_long_variable_name is not None and \
 very_long_variable_name.field > 0 or \
 very_long_variable_name.is_debug:
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
```

```
if (
    very_long_variable_name is not None
    and very_long_variable_name.field > 0
    or very_long_variable_name.is_debug
):
    z = "hello " + "world"
else:
    world = "world"
    a = "hello {}".format(world)
    f = rf"hello {world}"
```

❑ VSCode에서 포매터로 지정하기

- Black을 기본 포맷터로 실행하도록 VSCode 기본 설정에 추가
  - .vscode/setiings.json 수정
  - Ctrl + Shift + P 로 명령 팔레트 이동
  - 다음 항목 추가

```
"python.formatting.provider": "black“, # VSCode의 기본 포맷터 대신 Black을 사용하게 해주는 설정
"editor.formatOnSave": true # 코드를 저장할 때 마다 자동으로 포맷팅 해주는 설정
```

❑ 파일 저장하기

- calc.py 파일의 저장 전 후의 차이

```
PS C:\Users\djhan\Prj> black --check .
would reformat apitemp.py
would reformat calc.py
would reformat testSum.py

Oh no! 💥 💔 💥
3 files would be reformatted, 2 files would be left unchanged.
PS C:\Users\djhan\Prj> black --check .
would reformat apitemp.py
would reformat testSum.py

Oh no! 💥 💔 💥
2 files would be reformatted, 3 files would be left unchanged.
```

Black의 포맷팅 항목

❑ 별도 자료 제공

2 단위 테스트와 커버리지

Python의 단위 테스트

Synetics

❑ 단위 테스트

- 보통 메소드/함수를 대상으로 함
- "이런 값을 입력하고 실행하면 아마 이런 출력값이 나올텐데, 정말 예상대로 출력되었는가?" 확인
- 리팩토링의 기본 조건
- 일반적으로 단위 테스트를 도와주는 단위 테스트 프레임워크 활용

❑ Python의 단위 테스트 프레임워크

- unittest: Python에 기본으로 포함된 단위 테스트 프레임워크
- Pytest: 별도의 설치가 필요하며, unittest를 포함하여 테스트 기능을 확장한 프레임워크

unittest의 기본 사용 방법

Synetics

❑ 테스트 작성 기본

- unittest를 import
- Test Class를 만들고 unittest.TestCase 상속
- 관례적으로, test_ 를 붙인 테스트 함수 작성

❑ Assertion

- 함수의 매개변수(parameter)와 반환값(return value)을 이용하여 테스트
- 특정 함수에 매개변수를 지정하고 실행하였을 때,
  반환값이 개발자가 예상한 값과 같은지 확인하는 방법

Assertion 함수	내용
assertEqual(a, b)	기대값과 결과값이 동일한지 비교
assertNotEqual(a, b)	기대값과 결과값이 동일하지 않은지 비교
assertTrue(x)	기대값이 ture 인지 비교
assertFalse(x)	기대값이 false 인지 비교

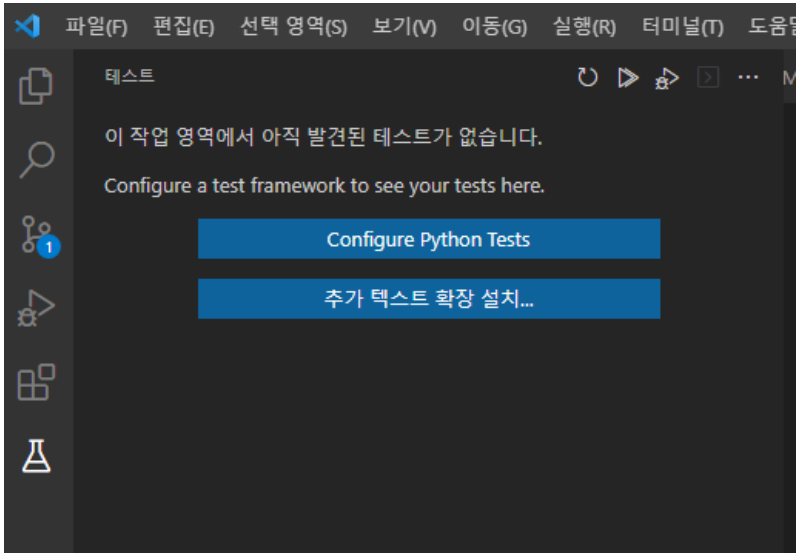
테스트 실행 방법

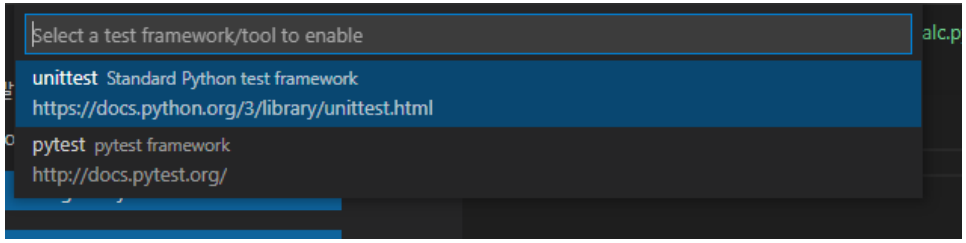
❑ 명령줄 실행

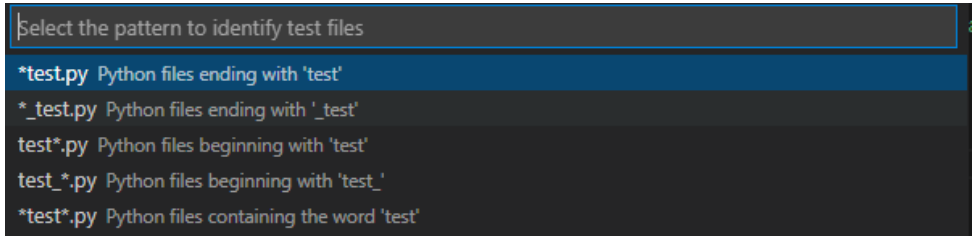
- python –m unittest // 전체 테스트 실행
- python –m 모듈명 // 해당 모듈만 테스트 실행
  - 예) python –m testSum

❑ VSCode에서 실행 설정

- 좌측 메뉴에서 "테스트 " 선택
- Configure Python Test 선택 후, unittest 선택
- 테스트 파일 이름 유형에, test_*.py 선택







unittest 예제

더하기 함수

```python
def sum(a, b):
    return a + b
```

테스트 클래스

```python
import unittest
import calc


class sumtest(unittest.TestCase):
    def test_Sum(self):
        result = calc.sum(3, 5)
        self.assertEquals(8, result)
```

테스트 결과

```
PS C:\Users\djhan\Prj> python -m unittest testSum
C:\Users\djhan\Prj\testSum.py:8: DeprecationWarning: Please use assertEqual inst
ead.
  self.assertEquals(8, result)
.
----------------------------------------------------------------------
Ran 1 test in 0.002s
```

기능이 추가된 계산기 클래스

Synetics

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



def minus(a, b):
    return a - b
```

기능이 추가된 계산기 테스트 클래스

```
import unittest
import calc



class sumtest(unittest.TestCase):
    def testSum(self):
        result = calc.sum(3, 5)
        self.assertEqual(8, result)

    def testMinus(self):
        result = calc.minus(5, 3)
        self.assertEqual(2, result)
```

Mock을 활용하기

Synetics

❑ Mock이란?

- 테스트의 의존성을 줄이기 위한 가짜 객체를 테스트에 활용하는 방법
- 테스트 의존성?
  - 단위 테스트에서 DB에 항상 접속해야 한다면
  - DB에서 외부 API를 항상 불러야 한다면
- 특정 함수가 호출될 때, 특정 값으로 지정하는 기능
  - 실제 반환 값이 얼마이든, 내가 지정한 값으로 고정
  - 예) 외부 API에서 불러온 값이 무조건 2로 고정

❑ Python의 Mock 기능

- Unittest의 MagicMock 사용 가능
- 테스트 함수에서 @patch Annotation을 사용해서 지정
  - 예1) @patch("calc.api", return_value=2) // calc의 api가 호출되면 반환값을 2로 고정해라
  - 예2) @patch("calc.api") // calc의 api가 Mock 대상이다

```
def testsum(self, mock):  // mock이 포함된 함수 선언
    mock.return_value = 2  // calc의 api가 호출되면 반환값을 2로 고정해라
```

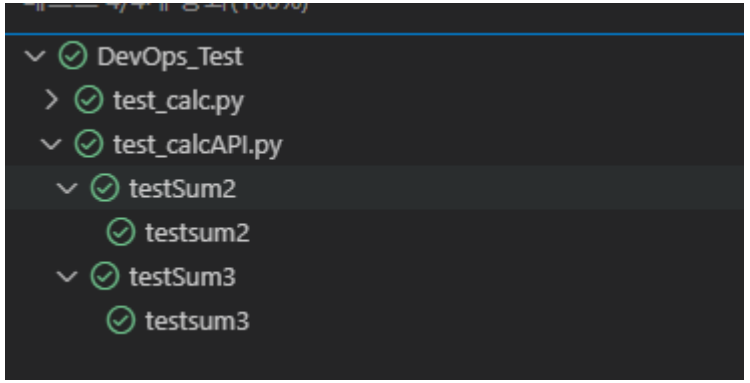
외부 API 연동이 추가된 계산기 클래스

Synetics

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


def minus(a, b):
    return a - b


def sumWithApi(a):  #api 호출
    return a + api()


def api():  # api
    return 3
```

외부 API 연동을 Mock 처리한 테스트 클래스

```
import unittest
from unittest.mock import MagicMock, patch
import calc



class testSum2(unittest.TestCase):
    @patch("calc.api", return_value=2)
    def testsum2(self, mock):

        result = calc.api()
        self.assertEqual(result, 2)



class testSum3(unittest.TestCase):
    @patch("calc.api")
    def testsum3(self, mock):

        mock.return_value = 2
        result = calc.sumWithApi(5)
        self.assertEqual(result, 7)
```



Coverage

Synetics

❑ 소스코드의 얼마만큼 테스트 했는가?

- 라인(Line) 커버리지 (Statement 커버리지 라고도 함)
- 브랜치(Branch) 커버리지 (Decision 커버리지 라고도 함)

❑ 왜 소스코드 테스트 커버리지가 중요한가?

- 소스코드 품질을 위한 테스트 목표 (QA가 닥달하기 딱 좋은 지표)
  - 안전성 분야에서는 100%가 기준
- 테스트가 부족한 부분은 확인
  - 불필요한 코드인가?
  - 테스트가 충분한가?

❑ Python의 경우, 기본적으로 라인 커버리지만 측정

Line 커버리지

❑ Line(Statement) 커버리지 산출 공식

- 테스트한 구문 / 전체 구문

※ 구문(statement): ';'로 끝나는 명령 라인

❑ 의미

- 프로그래밍 언어의 기본 구성 단위에 대한 검증
- 개발자가 구현한 전체 구문 및 블록에 대하여 최소한의 검증 수행
- 낮은 단계의 커버리지이나 시스템이 복잡할 수록 달성하기 어려움

❑ 예제

```
#include <stdio.h>

void api1(int i){

        printf("...₩n");

        if(i<0){
                printf("...₩n");
                printf("...₩n");
        }
}
```

Statement Coverage

Q1) i를 0으로 테스트 한 경우:

Q2) i를 -1로 테스트 한 경우:

기혼자이고

나이가 35세를 초과하면

100만원의 소득 공제를 받음

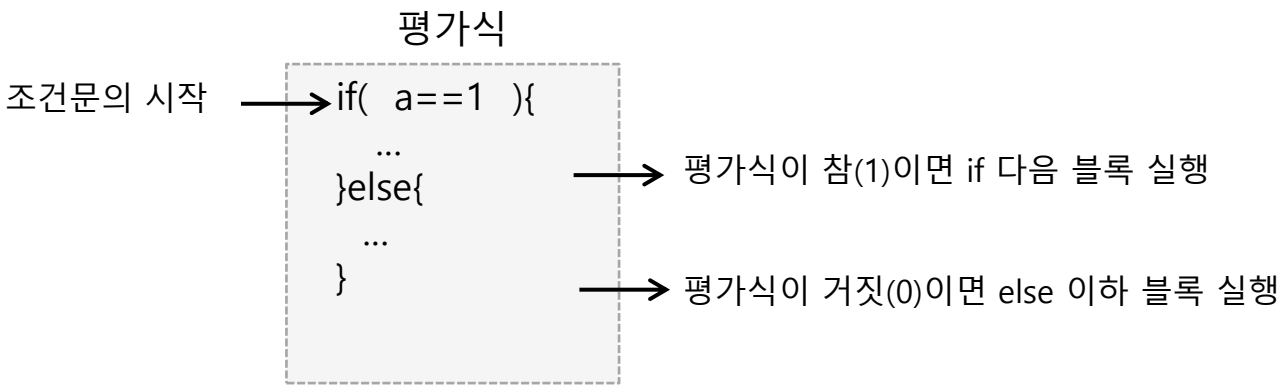
Condition Condition

```
if(married && (age > 35))
  then amount = amont + 100;
```

Decision

Branch(Decision) 커버리지

- ✓ **Decision Coverage 산출 공식**
  - 테스트한 Decision 수 / 전체 Decision 수
  - Decision Coverage 100% 달성하기 위한 최대 테스트 케이스 수
    - 전체 조건문 수 * 2
- ✓ **의미**
  - 결정의 참/거짓을 한 번씩 테스트 해보는 것
- ✓ **코드 분석 팁(Code Reading Tip)**



  - 각 조건문은 2개의 분기를 가진다.

Branch(Decision) 커버리지

```
void util(int a){
        if( a>0 ){
          ……
        }

        if(  a==1  ){
           …
        }else{
          …
        }
}
```

Q1) a를 0으로 테스트 한 경우:

Q2) a를 1으로 테스트 한 경우:

Q3) Decision Coverage 100%를 달성하기 위한 최소개의 테스트 케이스는?

도구 설치 및 실행

❑ pip install coverage

- 단, 설치 후 path에 pip의 패키지 등록 위치를 path에 추가하는 것을 권고
- 일반적으로 다음과 같음
  - C:₩Users₩사용자계정₩AppData₩Roaming₩Python₩Python310₩Scripts

❑ 커버리지 측정 실행

- 측정

  coverage run -m unittest discover

```
PS C:\Users\devops\DevOps_Test> coverage run -m unittest discover
C:\Users\devops\DevOps_Test\test_calc.py:12: DeprecationWarning: Pl
  self.assertEquals(2, result)
..C:\Users\devops\DevOps_Test\test_calcAPI.py:11: DeprecationWarnin
g: Please use assertEqual instead.
  self.assertEquals(result, 2)
..
----------------------------------------------------------------------
---
Ran 4 tests in 0.031s
```

커버리지 측정 확장

❑ 커버리지 결과 보기(터미널)

- coverage report

```
PS C:\Users\devops\DevOps_Test> coverage report
Name              Stmts   Miss  Cover
-----------------------------------------
calc.py               8      1    88%
test_calc.py          9      0   100%
test_calcAPI.py      14      0   100%
-----------------------------------------
TOTAL                31      1    97%
```

❑ 커버리지 결과 보기(html)

- coverage html
- 결과 저장 폴더가 생성되고, html 파일이 생성

Coverage report: 97%

Module	*statements*	*missing*	*excluded*	*coverage*
calc.py	8	1	0	88%
test_calc.py	9	0	0	100%
test_calcAPI.py	14	0	0	100%
Total	**31**	**1**	**0**	**97%**

coverage.py v6.3.1, created at 2022-02-17 06:33 +0900

Jenkins에서 결과 보기

Synetics

❑ Jenkins에서 결과를 확인하기 위해 다음이 필요

- coverage 결과를 xml로 출력하기
  - coverage xml –o coverage.xml
  - coverage.xml 로 결과를 출력하라는 명령
- 빌드 후 조치 추가해서, coverage 표현 지정

Coverage 측정 및 표현 설정 - 1

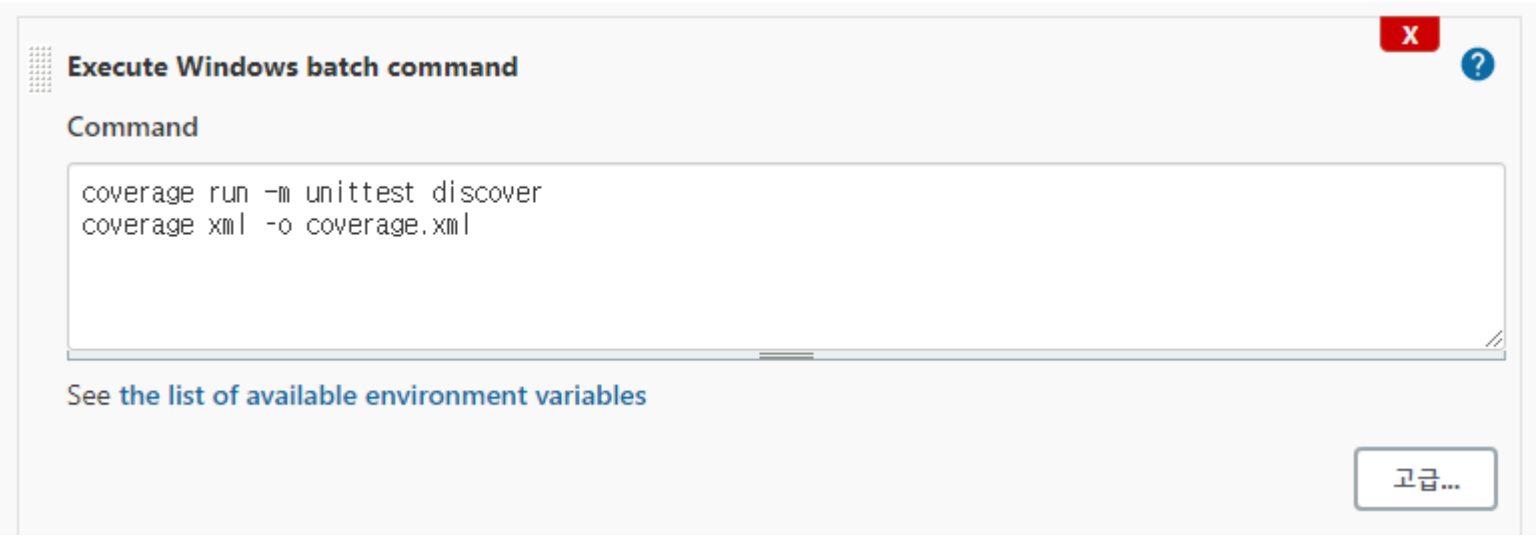
Synetics

❑ Jenkins 서버의 Coverage 실행 설정

- Jenkins 서버도 coverage 설치 및 path 설정

❑ Jenkins Job에서 Coverage 실행 설정

- coverage run -m unittest discover
- coverage xml –o coverage.xml

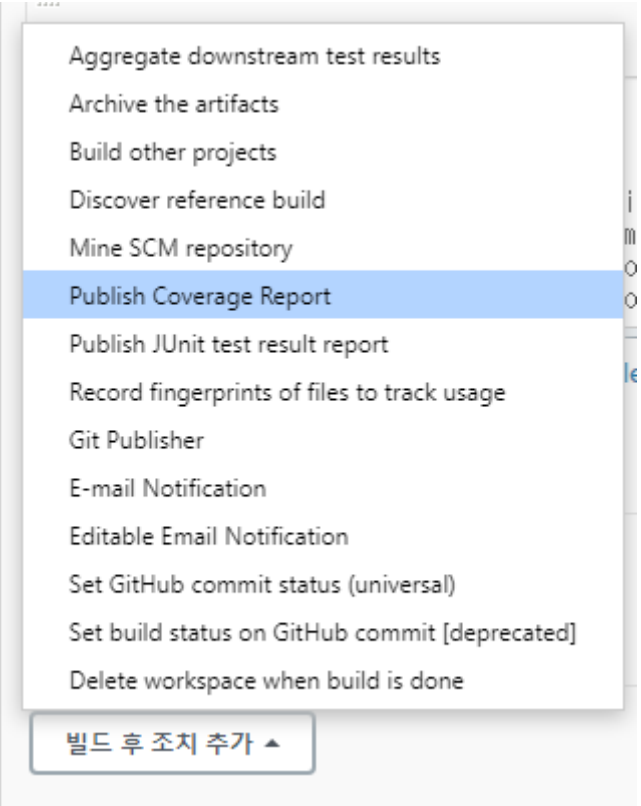


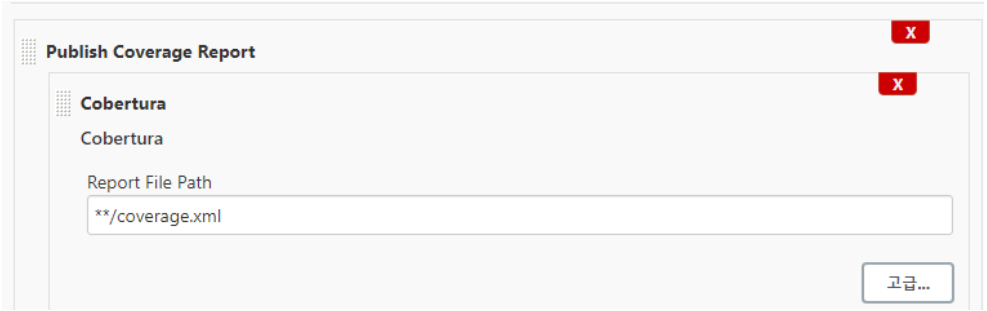
* 단, 처음 실행 시 pip install coverage 필요

Coverage 측정 및 표현 설정 - 2

❑ Jenkins Job에서 커버리지 표현 설정

- 빌드 후 조치 추가에서, Publish coverage report 선택
- Add를 클릭해서, Cobertura 선택
- Publish coverage report 에서 앞서 설정한 xml 파일명 지정
  - **/coverage.xml





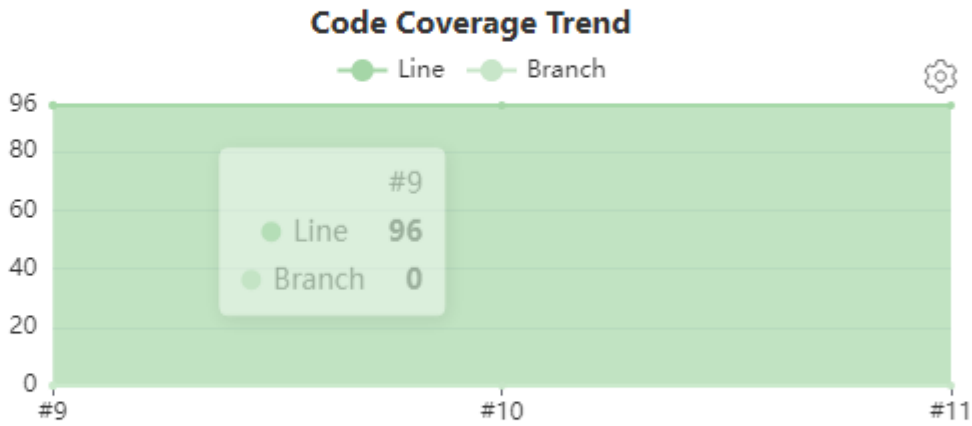
Jenkins에서 커버리지 확인 - 추세

Project PythonTest

설명을 작성합니다.

내용 수정

프로젝트 중지하기



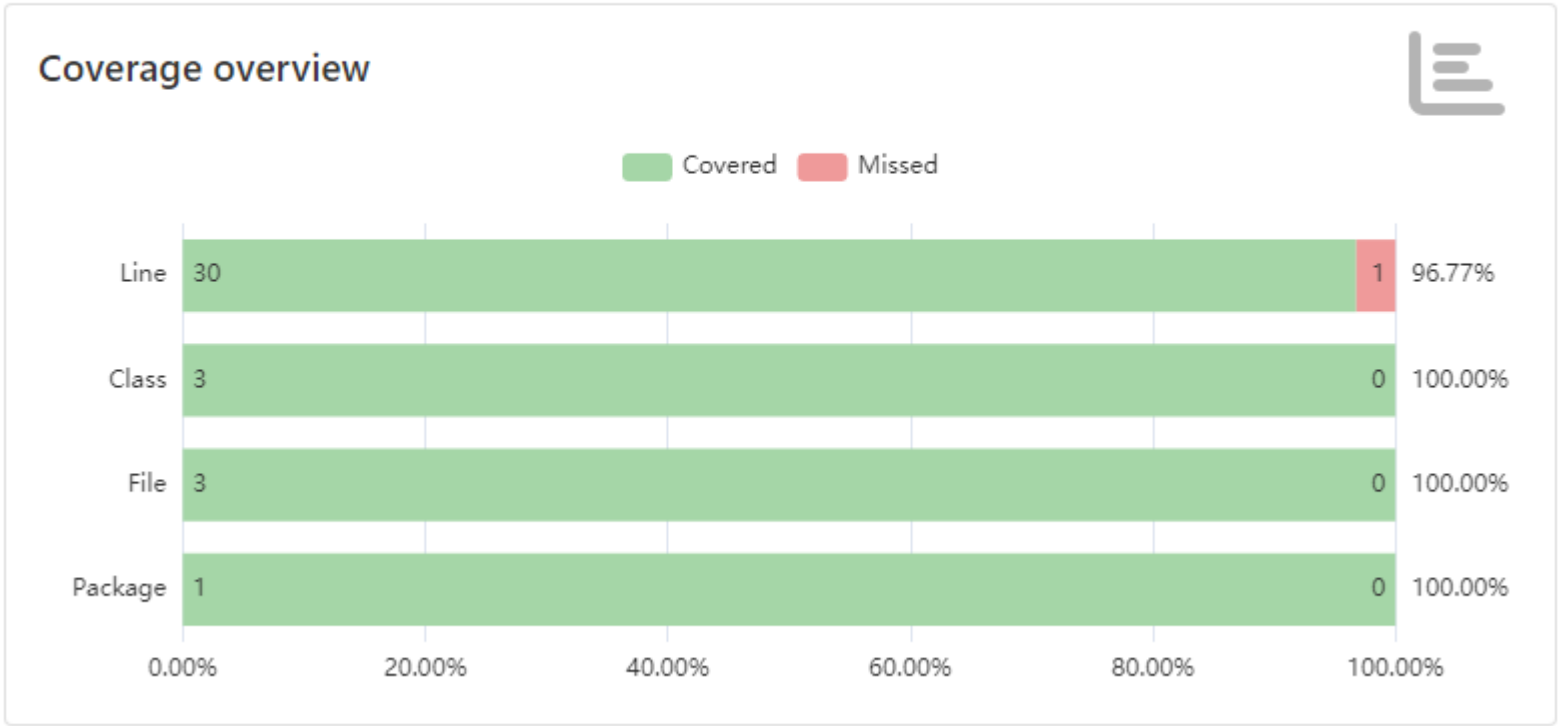
작업 공간

최근 변경사항

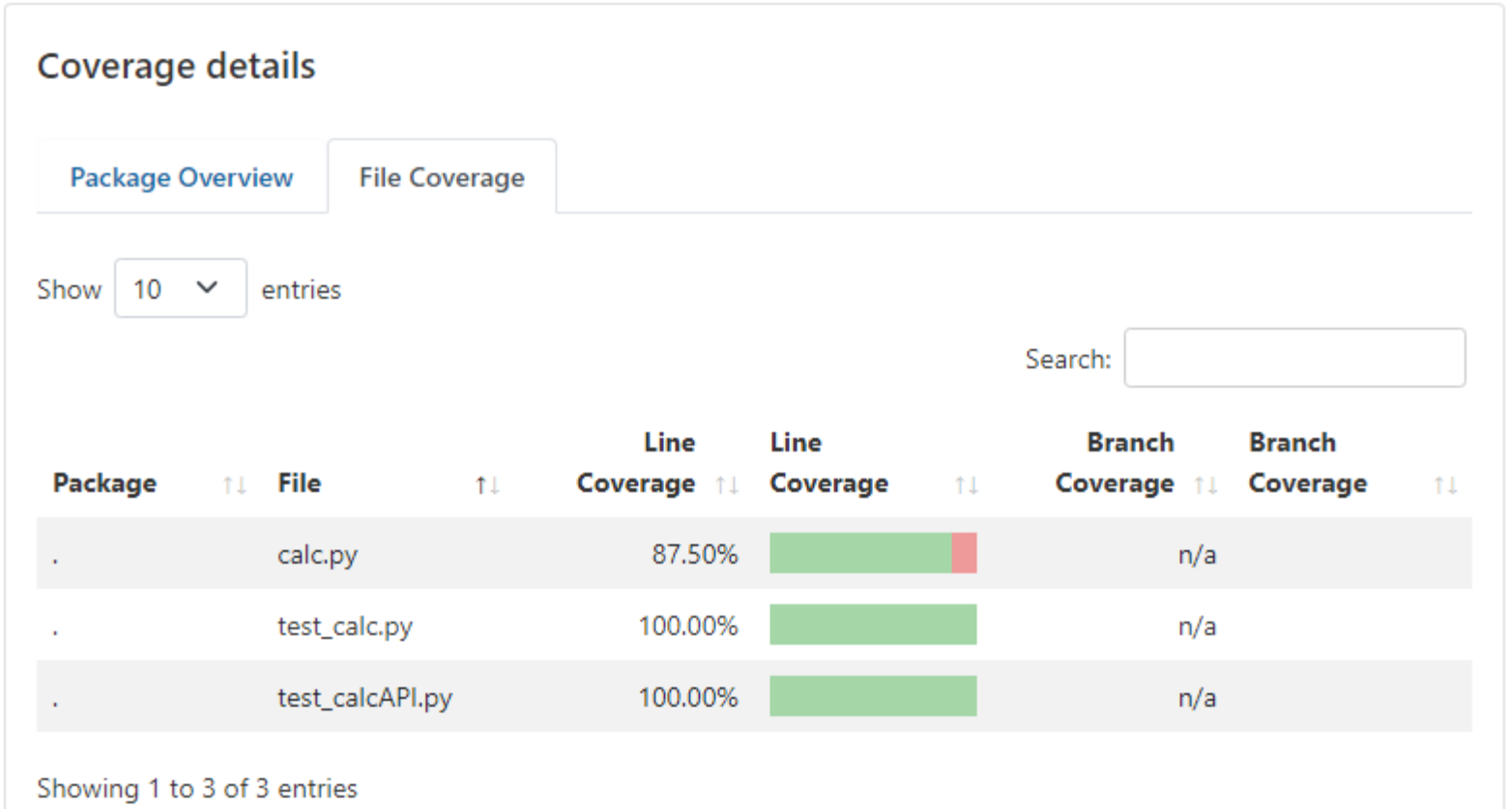
고정링크

- Last build, (#11),1 min 13 sec 전
- Last stable build, (#11),1 min 13 sec 전
- Last successful build, (#11),1 min 13 sec 전
- Last failed build, (#6),28 min 전
- Last unsuccessful build, (#6),28 min 전
- Last completed build, (#11),1 min 13 sec 전

Jenkins에서 커버리지 확인 - 전체



Jenkins에서 커버리지 확인 - 파일별



Coverage details

Package Overview | File Coverage

Show 10 entries

Search:

Package	File	Line Coverage	Line Coverage	Branch Coverage	Branch Coverage
.	calc.py	87.50%		n/a	
.	test_calc.py	100.00%		n/a	
.	test_calcAPI.py	100.00%		n/a	

Showing 1 to 3 of 3 entries

End of Document

Synetics

Any Question?