



TDD 해보기 : 은행계좌 KATA

TDD 기본 예제, TDD 한번 더 해보기

기능

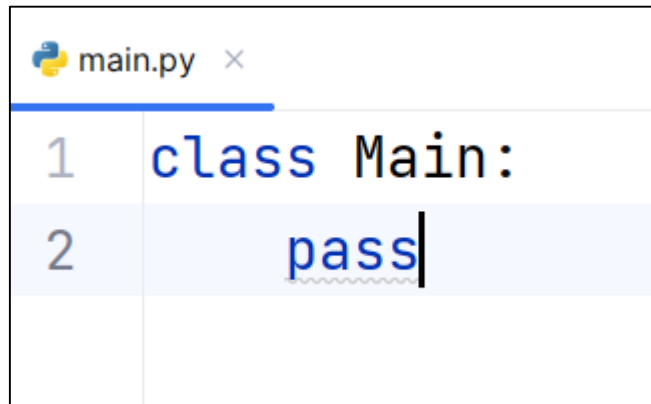
1. 입금 / 출금
2. 잔고 조회

실패하기 단계에서는...

- 내가 무엇을 할지 고민하고, 내가 구현할 기능들을 적어 둔다.
- 클라이언트가 사용하는 인터페이스를 고민한다.
 - 작성하고자 하는 메서드 이름을 결정한다.
 - 함수에 Input(Parameter)값과 Output(Return) 값을 결정한다.
- 스켈레톤 코드를 구현하고 시작해도 좋다.

main.py 기본 코드를 작성한다.

main.py 파일 생성 후 기본 코드 작성

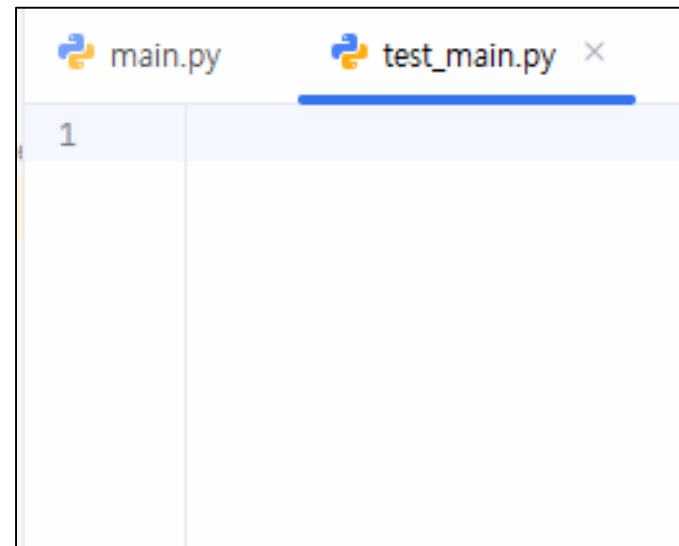
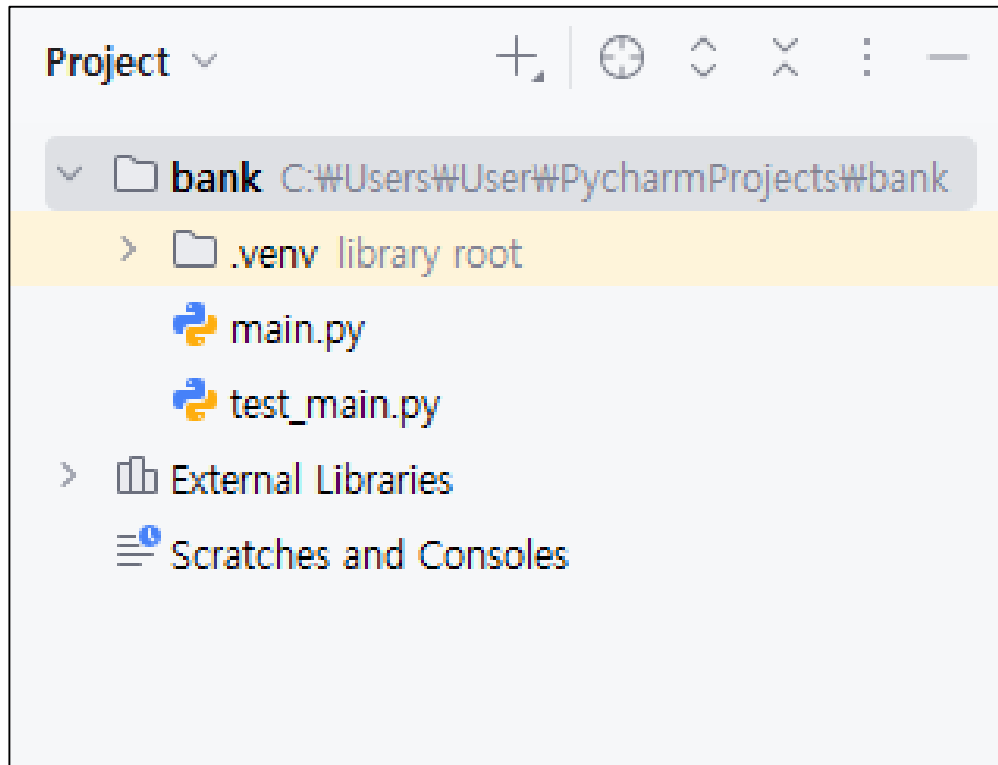


```
main.py x
1 class Main:
2     pass
```

The image shows a code editor window titled 'main.py' with a close button. The editor contains two lines of Python code. Line 1 is 'class Main:' and line 2 is 'pass'. The word 'pass' is underlined with a wavy line, indicating it is a placeholder for code. The cursor is at the end of the 'pass' line.

Test File 준비

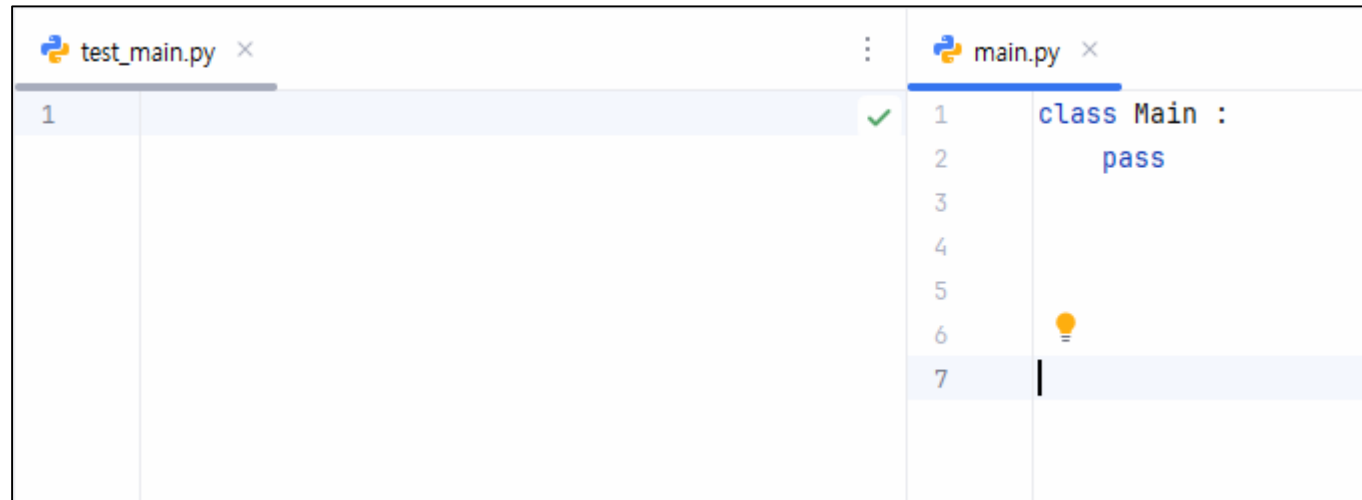
test_main.py 파일을 준비



창 배치하기

왼쪽 : test_main.py

오른쪽 : main.py



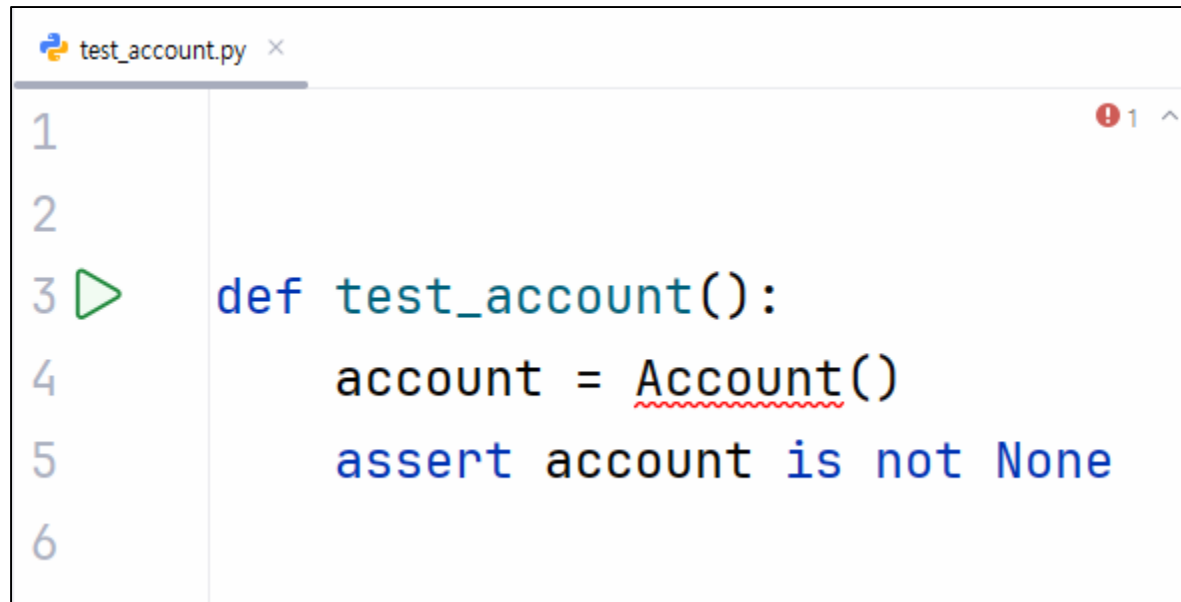
Step 1. 어떤 클래스를 만들까?

Red

Red 은 ToDo 이다.

Account 라는 Class를 만들자.

Red 단계는 내가 지금 당장 할 일(ToDo)를 기록하는 것이다.

A screenshot of a code editor window titled 'test_account.py'. The editor shows a Python function 'def test_account():' with two lines of code: 'account = Account()' and 'assert account is not None'. The 'Account' class name is underlined with a red squiggly line, indicating a missing definition. A green play button icon is next to line 3. A red circle with the number '1' and an upward arrow is in the top right corner of the editor area.

```
1  
2  
3 def test_account():  
4     account = Account()  
5     assert account is not None  
6
```

빌드가 안되어도, Fail 단계가 맞다.

왼쪽 : test_main.py → test_account.py로 이름을 바꾸자.

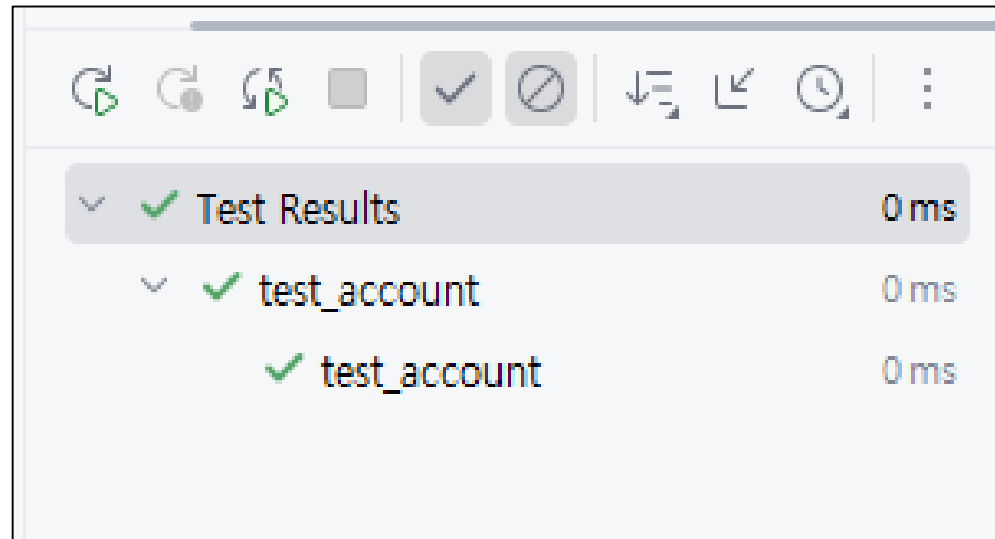
오른쪽 : main.py → account.py , Account 클래스명 변경

test_account.py ×	account.py ×
1 from account import Account ✓	1 class Account:
2	2 pass
3	3
4 ▶ def test_account():	
5 account = Account()	
6 assert account is not None	
7	
8	
9	

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

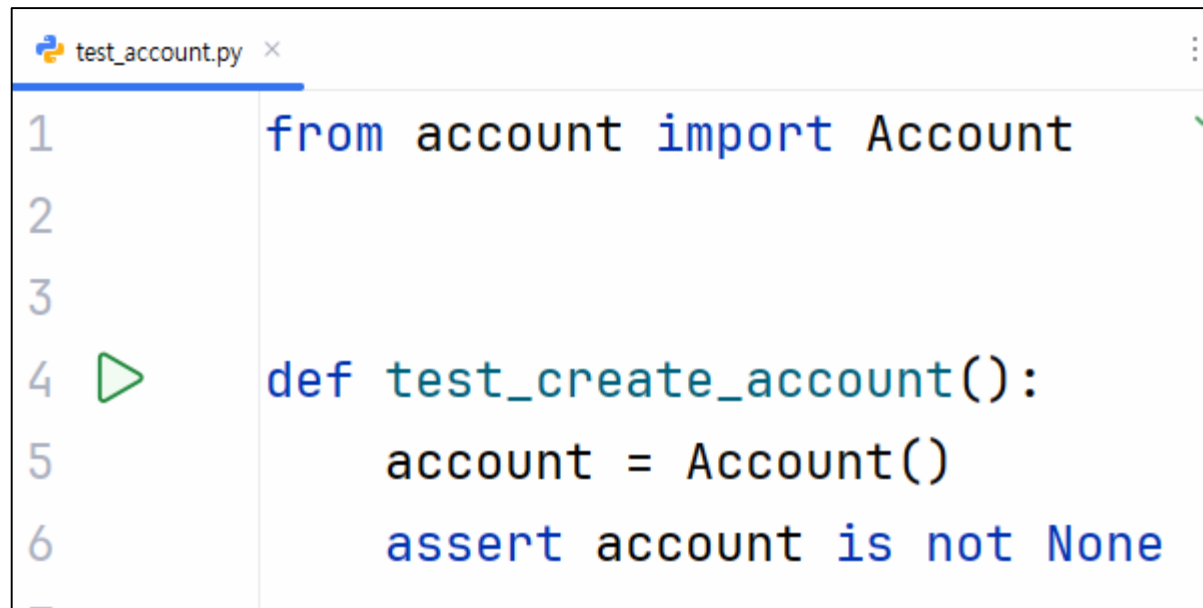


Refactor 단계 이다.

이제 더 깔끔한 코드를 위해 고민을 한다.

할 것이 없다면,

이제 다음 Red(ToDo) 단계로 넘어간다.



```
test_account.py x
1 from account import Account
2
3
4 def test_create_account():
5     account = Account()
6     assert account is not None
```

Test Method 이름만 바꿔주었다.

Step 2. 할일 정하기

Red

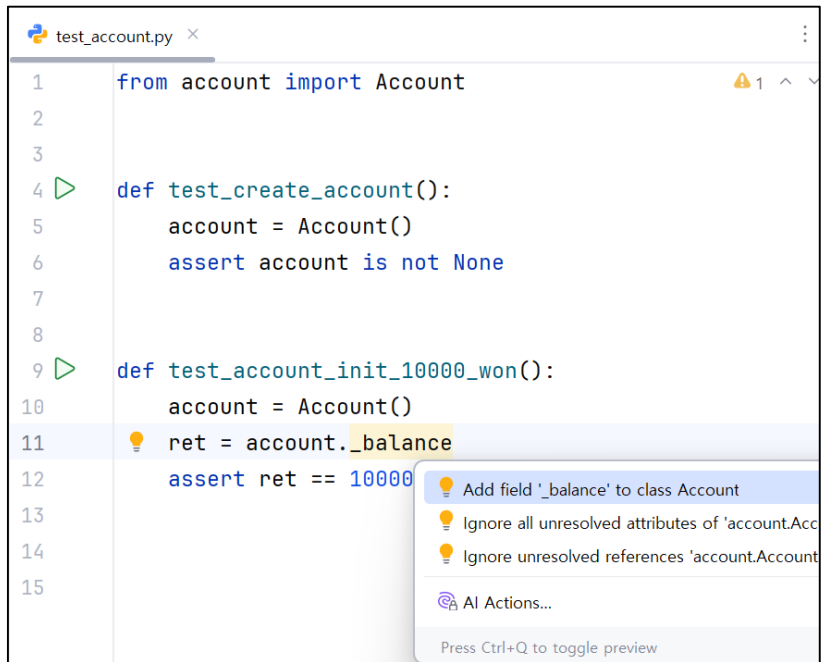
계좌를 생성시,
10000원 생성되고 balance 로 현재 계좌 값을 확인할 수 있다.

```
test_account.py ×
1  from account import Account
2
3
4  ▶ def test_create_account():
5      account = Account()
6      assert account is not None
7
8
9  ▶ def test_account_init_10000_won():
10     account = Account()
11     ret = account._balance
12     assert ret == 10000
13
```

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)



```
test_account.py x
1 from account import Account
2
3
4 def test_create_account():
5     account = Account()
6     assert account is not None
7
8
9 def test_account_init_10000_won():
10     account = Account()
11     ret = account._balance
12     assert ret == 10000
13
14
15
```

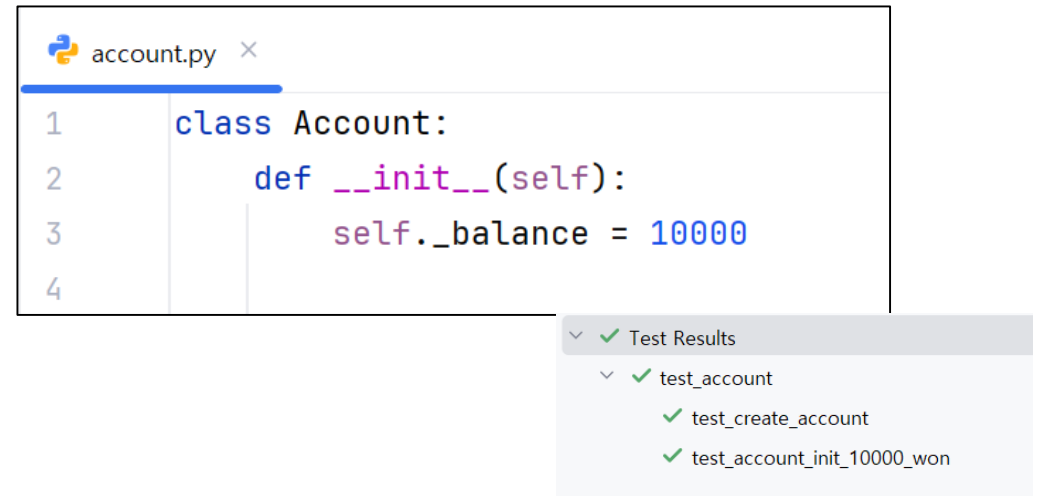
Lightbulb icon: Add field '_balance' to class Account

Lightbulb icon: Ignore all unresolved attributes of 'account.Account'

Lightbulb icon: Ignore unresolved references 'account.Account'

AI Actions...

Press Ctrl+Q to toggle preview



```
account.py x
1 class Account:
2     def __init__(self):
3         self._balance = 10000
4
```

Test Results

- ✓ test_account
 - ✓ test_create_account
 - ✓ test_account_init_10000_won

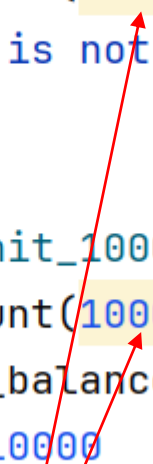
Step 3. 할일 정하기

Red

계좌 생성시, 초기 입금 비용을 결정하도록 한다.

Default 생성자를 막고, 초기 값을 입력하도록 한다.

```
def test_create_account():  
    account = Account(10000)  
    assert account is not None  
  
def test_account_init_10000_won():  
    account = Account(10000)  
    ret = account._balance  
    assert ret == 10000
```



기존 작성했던 Unit Test에 10000 을 대입한다.

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
def test_create_account():
    account = Account(10000)
    assert account is not None

def test_account_init_10000_won():
    account = Account(10000)
    ret = account._balance
    assert ret == 10000
```

Remove argument

Change the signature of `__init__(self)`

Change Signature

Name: `__init__`

+ - ↑ ↓

self

Name: `balance` Default value: `10000` Use default val...

Signature Preview

`__init__(self, balance)`

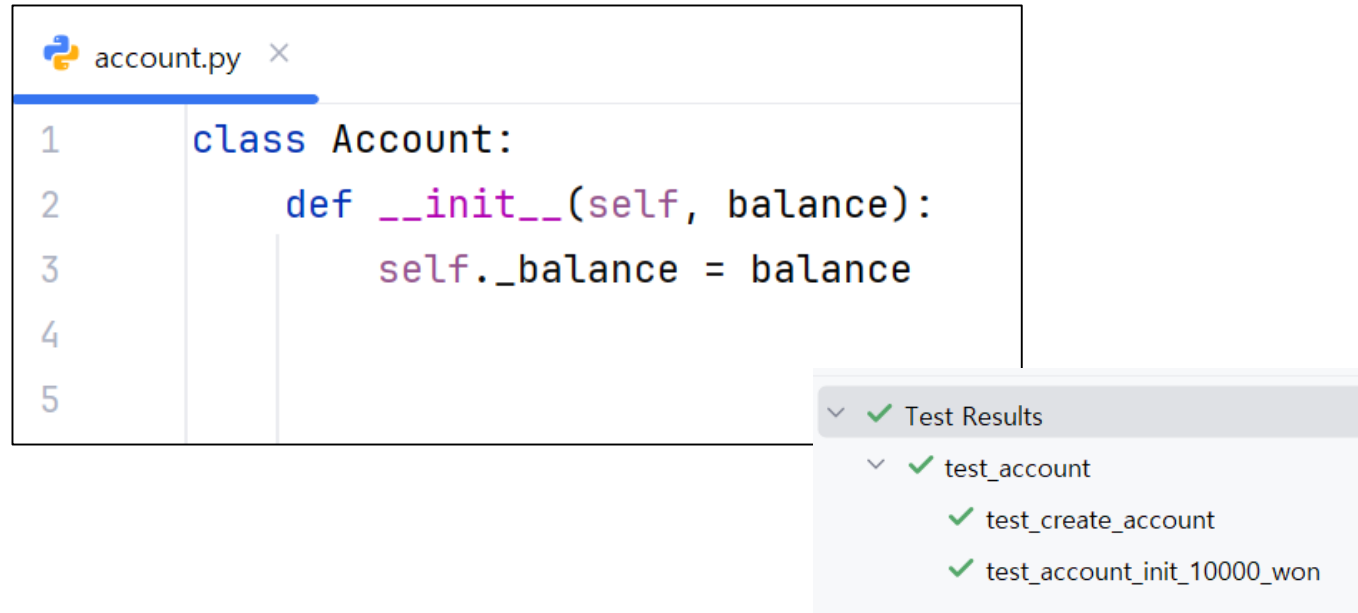
? Refactor Preview Cancel

```
class Account:
    def __init__(self, balance):
        self._balance = 10000
```

Test Results

- test_account
 - test_create_account
 - test_account_init_10000_won

하드코딩 되어 있는 10000 을 매개변수로 변경



The screenshot shows a code editor window titled 'account.py' with a close button. The code defines a class 'Account' with an '.__init__' method that takes 'self' and 'balance' as arguments and assigns 'self._balance = balance'. The code is displayed on five lines. To the right of the code editor, a 'Test Results' panel is visible, showing a list of test cases that have all passed successfully, indicated by green checkmarks.

```
1 class Account:
2     def __init__(self, balance):
3         self._balance = balance
4
5
```

Test Results

- test_account
 - test_create_account
 - test_account_init_10000_won

입금 구현하기

deposit(금액) : 현재 계좌에 금액을 추가한다.

```
def test_deposit_and_confirmation():  
    account = Account(10000)  
    account.deposit(500)  
    assert account._balance == 10500
```


빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
def test_deposit_and_confirmation():  
    account = Account(10000)  
    account.deposit(500)  
    assert account
```

💡 Add method deposit() to class Account

```
class Account:  
    def __init__(self, balance):  
        self._balance = balance  
  
    def deposit(self, param):  
        self._balance = 10500
```

✓ Test Results

- ✓ test_account
 - ✓ test_create_account
 - ✓ test_account_init_10000_won
 - ✓ test_deposit_and_confirmation

deposit 로직 일반화, money로 네이밍 변경
리팩토링 수행 후, Test를 한번 한다.

```
account.py x
1 class Account:
2     def __init__(self, balance):
3         self._balance = balance
4
5     def deposit(self, param):
6         self._balance += param
7
```

기존 self._balance = 10500 에서
10500 은 사실 self._balance + param 이다

```
account.py x
1 class Account:
2     def __init__(self, balance):
3         self._balance = balance
4
5     def deposit(self, money):
6         self._balance += money
7
```

Test Results

- test_account
 - test_create_account
 - test_account_init_10000_won
 - test_deposit_and_confirmation

출금하기 구현하기

withdraw(금액) : 현재 계좌에 금액을 뺀다.

```
def test_withdraw_and_confirmation():  
    account = Account(10000)  
    account.withdraw(600)  
    assert account._balance == 9400
```

빠르게 작업을 끝냈다.

Green은 빠르게 Red를 없애는 것이 중요하다.

고민은 크게 하지 않는다. (고민은 다음 단계에서)

```
def test_withdraw_and_confirmation():  
    account = Account(10000)  
    account.withdraw(600)  
    assert accou
```

💡 Add method withdraw() to class Account

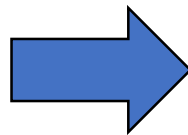
```
class Account:  
    def __init__(self, balance):  
        self._balance = balance  
  
    def deposit(self, money):  
        self._balance += money  
  
    def withdraw(self, param):  
        self._balance = 9400
```

✓ Test Results

- ✓ test_account
 - ✓ test_create_account
 - ✓ test_account_init_10000_won
 - ✓ test_deposit_and_confirmation
 - ✓ test_withdraw_and_confirmation

withdraw 로직 일반화, money로 파라미터 네이밍 변경

```
def withdraw(self, param):  
    self._balance = 9400
```



```
def withdraw(self, money):  
    self._balance -= money
```

✓ Test Results

- ✓ test_account
 - ✓ test_create_account
 - ✓ test_account_init_10000_won
 - ✓ test_deposit_and_confirmation
 - ✓ test_withdraw_and_confirmation

[참고] 로버트 C 마틴의 TDD 세 가지 Rule.

로버트 C 마틴의 세 가지 규칙

1. 실패하는 UnitTest를 작성할 때까지, Production Code를 작성하지 않는다.
2. **Compile은 실패하지 않으면서**, 실행이 실패하는 정도로만 Unit Test를 작성한다.
3. 현재 실패하는 UnitTest에 통과될 정도로만 실제 코드를 작성한다.

세 가지 규칙을 따르면 개발과 테스트가 대략 30초 주기로 묶인다.

이 방식대로 라면, 수천개에 달하는 Test Case가 나오지만, 관리 문제를 유발하기도 한다.

[도전] TDD로 다음 기능을 개발해본다.

1. 5% 복리 적용하기

5% 복리가 적용되어 금액이 올라가는 메서드

2. 은행 이자 Setter로 만들기

원하는 은행 이자로 지정하는 기능 추가.

3. 은행 이자율로 복리 적용하기

•1번에서 구현했던 기능을 수정하여 개발

4. n년 이후, 예상 복리 금액 알려주기 기능 추가.

•n년 동안 복리 이자를 적용하면 얼마가 되는지 알려주는 기능 추가하기