

# 세상에서 제일 쉬운 러스트 프로그래밍

## 14장. 테스트

윤인도

[freedomzero91@gmail.com](mailto:freedomzero91@gmail.com)

## 프로그래밍에서 테스트가 필요한 이유

- 소프트웨어 적응성을 확인하기 위해
- 오류 식별
- 추가 비용 방지
- 소프트웨어 개발 가속화
- 위험 방지

## 테스트의 종류

- 단위 테스트(unit test)는 함수와 같은 개별 코드 단위에 집중
- 통합 테스트(integration test)는 여러 소프트웨어 구성 요소가 함께 작동하는 방식을 테스트

# 유닛 테스트

## 소스코드 살펴보기

- 파이썬에서 유닛테스트를 위해서 사용하는 라이브러리로 `pytest` 패키지를 많이 사용
- 아래 명령어로 패키지를 설치

```
pip install pytest
```

`logic.py` 모듈을 다음과 같이 작성

- `play()` 함수는 입력받은 두 카드의 값을 비교해 첫 번째 카드의 승패 유무를 리턴
- `stop()` 함수는 무조건 에러를 발생시켜 프로그램을 종료

```
from typing import Optional

CARDS = ["Rock", "Scissors", "Paper"]

def play(card1: str, card2: str) -> Optional[bool]:
    assert card1 in CARDS, "Invalid card1"
    assert card2 in CARDS, "Invalid card2"
    if card1 == card2:
        return None
    return ((card1 == "Rock" and card2 == "Scissors")
            or (card1 == "Scissors" and card2 == "Paper")
            or (card1 == "Paper" and card2 == "Rock"))

def stop():
    raise Exception("stop!")
```

## test.py 모듈

- `@pytest.mark.parametrize` 는 테스트의 각 파라미터를 테스트 수행 중에 동적으로 변경
- 여기서 승, 무, 패 3가지를 테스트하는 함수 `test_win`, `test_draw`, `test_lose` 와 함께 함수 `stop()` 이 에러를 발생시키는지 검사하는 `test_stop` 까지 총 4개의 테스트가 존재

```

import pytest
from logic import play, stop

@pytest.mark.parametrize(
    "card1, card2",
    [("Rock", "Scissors"), ("Scissors", "Paper"), ("Paper", "Rock")],
)
def test_win(card1, card2):
    assert play(card1, card2) == True

@pytest.mark.parametrize(
    "card1, card2",
    [("Rock", "Rock"), ("Scissors", "Scissors"), ("Paper", "Paper")],
)
def test_draw(card1, card2):
    assert play(card1, card2) == None

@pytest.mark.parametrize(
    "card1, card2",
    [("Scissors", "Rock"), ("Rock", "Paper"), ("Paper", "Scissors")],
)
def test_lose(card1, card2):
    assert play(card1, card2) == False

def test_stop():
    with pytest.raises(Exception) as exc:
        stop()

```

## 테스트 실행

```
pytest test.py
```

## 실행 결과

```
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /ch10/python
plugins: dash-2.0.0, anyio-3.3.4
collected 10 items

test.py ..... [100%]

===== 10 passed in 0.05s =====
```

## lib.rs 로 라이브러리 크레이트 생성

```
#[derive(PartialEq)] // 열거형끼리의 대소비교
pub enum Cards {
    Rock,
    Scissors,
    Paper,
}

/// Demonstrate Rock, Scissors, Paper
/// ```
/// use rust_part::{play, Cards};
///
/// let result = play(Cards::Rock, Cards::Scissors);
/// assert_eq!(result, Some(true));
/// ```
pub fn play(card1: Cards, card2: Cards) -> Option<bool> {
    if card1 == card2 {
        return None;
    }
    match (card1, card2) {
        (Cards::Rock, Cards::Scissors) => Some(true),
        (Cards::Scissors, Cards::Paper) => Some(true),
        (Cards::Paper, Cards::Rock) => Some(true),
        _ => Some(false),
    }
}

pub fn stop() {
    panic!("stop!");
}
```



## 파라미터화된 테스트는 기본적으로 지원하지 않음

```
#[cfg(test)]
pub mod test {
    use super::*;
    #[test]
    fn test_win() {
        assert_eq!(play(Cards::Paper, Cards::Rock), Some(true));
        assert_eq!(play(Cards::Scissors, Cards::Paper), Some(true));
        assert_eq!(play(Cards::Paper, Cards::Rock), Some(true));
    }
    #[test]
    fn test_draw() {
        assert_eq!(play(Cards::Rock, Cards::Rock), None);
        assert_eq!(play(Cards::Scissors, Cards::Scissors), None);
        assert_eq!(play(Cards::Paper, Cards::Paper), None);
    }
    #[test]
    fn test_lose() {
        assert_eq!(play(Cards::Rock, Cards::Paper), Some(false));
        assert_eq!(play(Cards::Paper, Cards::Scissors), Some(false));
        assert_eq!(play(Cards::Scissors, Cards::Rock), Some(false));
    }

    #[test]
    #[should_panic(expected="stop!")]
    fn test_stop(){
        stop();
    }
}
```

- 러스트는 테스트 파일을 별도로 만들지 않고, 같은 파일 안에 `test` 모듈을 넣어서 작성
- 테스트 모듈에서 대상 모듈에 대한 접근이 용이 (비공개도 접근 가능)

아래 명령어로 테스트 모듈의 테스트들을 수행합니다.

```
cargo test
```

## 실행 결과

```
running 4 tests
test test::test_draw ... ok
test test::test_lose ... ok
test test::test_stop - should panic ... ok
test test::test_win ... ok
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
Doc-tests rust_part
running 1 test
test src/lib.rs - play (line 10) ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.59s
```

## 클래스와 구조체 테스트

logic.py 에 다음과 같은 클래스를 정의

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age

    def hi(self):
        return f"Hi, I'm {self.name}, I'm {self._age} years old."
```

테스트 모듈에서는 객체화를 한 다음 프로퍼티와 메서드가 잘 적용되는지를 테스트

```
def test_hi():  
    name = "John"  
    age = 30  
    person = Person(name, age)  
    assert person.hi() == f"Hi, I'm {name}, I'm {age} years old."  
    assert person.hi() == f"Hi, I'm {person.name}, I'm {person.age} years old."
```

- 러스트에서는 `lib.rs` 에 다음 구조체를 선언
- 먼저 `person` 모듈을 선언하고 그 다음 `Person` 구조체와 메서드를 정의

여기서 별도로 `person` 모듈을 만들면 테스트 모듈과의 구분이 쉬워짐!

```
pub mod person {
    pub struct Person {
        pub name: String,
        age: u8,
    }
    impl Person {
        pub fn new(name: &str, age: u8) -> Person {
            Person {
                name: name.to_string(),
                age: age,
            }
        }
        pub fn hi(&self) -> String {
            format!("Hi, I'm {}, I am {} years old.", self.name, self.age())
        }
        pub fn age(&self) -> u8 { self.age }
    }
}
```

그 다음 테스트 모듈에 아래 코드를 작성

```
#[cfg(test)]
pub mod test {
    use super::person;
    #[test]
    fn test_hi() {
        let name = "John";
        let age: u8 = 30;
        let person = person::Person::new(name, age);
        assert_eq!(
            person.hi(),
            format!("Hi, I'm {}, I am {} years old.", name, age)
        );
        assert_eq!(
            person.hi(),
            format!("Hi, I'm {}, I am {} years old.", person.name, person.age())
        );
    }
}
```

## 테스트 실행 결과

```
Running unittests src/lib.rs (target/debug/deps/rust_part-f0f4dd89a6954739)
  running 1 test
    test test::test_hi ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

## 비동기 함수 테스트

`#[tokio::test]` 를 함수에 추가하면 끝!

```
async fn give_order(order: u64) -> u64 {
    println!("Processing {order}...");
    tokio::time::sleep(std::time::Duration::from_secs(3 - order)).await;
    println!("Finished {order}");
    order
}

#[cfg(test)]
mod tests {
    use super::*;

    #[tokio::test]
    async fn test_give_order() {
        let result = give_order(1).await;
        assert_eq!(result, 1);
    }
}
```



## 실행 결과

```
running 1 test  
test tests::test_give_order ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 2.00s
```

## 문서 테스트

아까 `cargo test` 를 실행했을 때, 유닛 테스트 외에도 한 가지 테스트가 더 추가로 실행됐었습니다. 바로 문서가 잘 작성되었는지를 테스트하는 문서 테스트입니다.

```
Doc-tests rust_part

running 1 test
test src/lib.rs - play (line 10) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.55s
```

**play** 함수 위의 주석을 보면 다음과 같은 부분이 들어있습니다.

```
/// Demonstrate Rock, Scissors, Paper
///
/// ```
/// use rust_part::{play, Cards};
///
/// let result = play(Cards::Rock, Cards::Scissors);
/// assert_eq!(result, Some(true));
/// ```
```

- `/` 을 3개 달아서 함수에 해당하는 주석이라는 것을 표시
- 가장 윗 줄은 어떤 함수인지를 설명
- 그 다음 "````" 로 묶인 부분은 이 `play` 함수를 사용하기 위한 예제 코드
- 문서 테스트가 실행되면 이 예제 코드가 컴파일되는지를 테스트

```

/// A module for Person struct.
pub mod person {
    /// Person struct with name and age.
    /// ```
    /// use rust_part::person::Person;
    ///
    /// let person = Person::new("John", 30);
    /// person.hi();
    /// ```
    pub struct Person {
        pub name: String,
        age: u8,
    }
    /// Methods defined for Person struct.
    impl Person {
        pub fn new(name: &str, age: u8) -> Person {
            Person {
                name: name.to_string(),
                age: age,
            }
        }

        pub fn hi(&self) -> String {
            format!("Hi, I'm {}, I am {} years old.", self.name, self.age())
        }


        pub fn age(&self) -> u8 {
            self.age
        }
    }
}

```

그 다음 작성된 문서를 브라우저에서 확인하기 위해 아래 명령어를 실행합니다.

```
cargo doc --open
```

브라우저가 실행되고 아래와 같은 메인 페이지가 나타납니다.



## Crate rust\_part

Version 0.1.0

[All Items](#)


[Modules](#)


[Enums](#)

[Functions](#)

**Crates**

[rust\\_part](#)

 Click or press 'S' to search, '?' for more options...

[?](#) 

### Crate rust\_part

[source](#) · [\[-\]](#)

#### Modules

---

[person](#) A module for Person struct.

#### Enums

---

#### Cards

---

#### Functions

---

[play](#) Demonstrate Rock, Scissors, Paper

[stop](#)

여기서 `person::Person` 구조체로 들어가 보겠습니다.

## Struct `rust_part::person::Person`

[source](#) · [\[-\]](#)

```
pub struct Person {  
    pub name: String,  
    /* private fields */  
}
```

[\[-\]](#) Person struct with name and age.

```
use rust_part::person::Person;  
  
let person = Person::new("John", 30);  
person.hi();
```

구조체 정의에 퍼블릭 필드만 나오는 것을 알 수 있습니다. 그리고 아까 작성한 예제 코드도 나타납니다. 이처럼 코드에 작성한 주석을 바로 문서로 만들 수 있는 것이 러스트의 큰 장점입니다.



마지막으로 문서 테스트만 실행하는 방법은 다음과 같습니다.

```
cargo test --doc
```

## 모킹

파이썬에서 다양한 모킹을 사용하기 위해 `pytest-mock` 플러그인이 자주 사용됨

```
pip install pytest-mock
```

유닉스 파일 시스템에서 파일을 지우는 코드를 모킹을 사용해 테스트해보면 다음과 같습니다.

```
import os

class UnixFS:
    @staticmethod
    def rm(filename):
        os.remove(filename)

def test_unix_fs(mock):
    mock.patch('os.remove')
    UnixFS.rm('file')
    os.remove.assert_called_once_with('file')
```

- 러스트에서의 모킹은 파이썬과는 매우 다르게 사용됨
- 일반적으로 `mockall` 과 `mockall_double` 크레이트를 사용하는데, 파이썬과 달리 객체를 직접 모킹할 수 없음

```
cargo add mockall mockall_double
```

## 폴더 구조

```
├── api.rs  
├── fs.rs  
└── main.rs
```

여기서 `mockall::automock` 은 테스트 코드에서 사용될 모킹된 `MockFS` 구조체를 자동으로 생성

```
// api.rs
#[allow(dead_code)]
mod fs_api {
    use std::fs;

    pub struct FS {}

    #[cfg_attr(test, mockall::automock)]
    impl FS {
        pub fn new() -> Self {
            Self {}
        }
        pub fn remove_file(&self, filename: &str) -> Result<(), std::io::Error> {
            fs::remove_file(filename)
        }
    }
}
```

- `mockall_double::double` 을 사용하면 일반 코드에서는 위에서 정의한 `FS` 구조체가, 테스트에서는 자동으로 생성된 `MockFS` 구조체가 사용됨
- `UnixFS` 구조체는 `FS` 구조체의 인스턴스를 사용하여 파일을 제거하는 `rm` 메서드를 정의

```
// fs.rs
use mockall_double::double;

#[double]
use crate::fs_api::FS;

pub struct UnixFS {}

impl UnixFS {
    pub fn rm(fs: &FS, filename: &str) -> Result<(), std::io::Error> {
        fs.remove_file(filename)
    }
}
```

## 테스트 모듈

- `use mockall::predicate::*;` 는 모킹한 함수 `expect_remove_file` 의 예상 입력을 찾을 수 있도록 하는 함수
- 입력으로 문자열 슬라이스 `"file"` 이 들어오는 경우에 이 모킹 함수가 작동

```
#[cfg(test)]
mod test {
    use super::*;
    use mockall::predicate::eq;
    #[test]
    fn test_remove_file() {
        let mut fs = FS::default();
        fs.expect_remove_file()
            .with(eq("file"))
            .returning(|_| Ok(()));
        UnixFS::rm(&fs, "file").unwrap();
    }
}
```



main.rs 파일

```
mod api;
mod fs;

#[double]
use api::FS;
use fs::UnixFS;
use mockall_double::double;

fn main() {
    let fs = FS::new();
    if let Err(e) = UnixFS::rm(&fs, "file") {};
    println!("Error: {}", e);
}
```

`cargo run`으로 실행하면 실제로 파일이 없어 에러가 발생하지만 `cargo test`는 모킹을 사용하기 때문에 문제없이 테스트가 통과

## 실행 결과

```
running 1 test
test test::test_remove_file ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```