# 11. Writing Automatic Testing

- 테스트 메커니즘을 설명함.
- 테스트란? 코드가 의도대로 기능하는지 검증하는 함수로 세 가지 동작을 수행함.
    - 필요한 데이터나 상태 설정
    - 테스트할 코드 실행
    - 의도한 결과가 나오는지 확인

## 11.1 테스트 작성 방법

테스트 함수를 작성하기 위해서는 함수 선언 전`#[test]`로 어노테이션해주어야 함.
테스트는 cargo test 명령어로 실행되며 이 명령을 실행하면 러스트는 속성이 표시된 함수를 실행하고 결과를 보고하는 테스트 실행 바이너리를 빌드함.

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

- 이 함수를 작성하고 `cargo test`를 실행하게 되면
- `test result ok`를 볼 수 있다.
    - 이는 테스트를 성공적으로 마쳤다는 뜻이고 다르게 나온다면 해당 테스트에 문제가 있는 거싱다.
- 만약 테스트에 실패하면 `Test failed`라고 나오며 어떤 줄에서 문제가 생겼는지 어떤 함수에서 문제가 생겼는지 알려준다.

## assert! 매크로로 결과 검사하기

- 어떤 조건이 `true`인지 확인하는 테스트를 작성할 때 사용하는 함수이다.
- `assert!`는 boolean 값을 인수로 전달받아 `true`값인 경우 아무 일도 일어나지 않고, `false`인경우 `panic!`매크로를 호출함.

## assert_eq!, assert_ne! 매크로를 이용한 동등 테스트

- `assert_eq!`은 두 인자가 같은지 확인함.
- `assert_ne!`은 두 인자가 틀린지 확인함.

## 추가적인 메세지 적어주기

```
pub fn greeting(name: &str) -> String {
        String::from("Hello!")
}
#[test]
fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
                result.contains("Carol"),
                "Greeting did not contain name, value was
`{}`",
                result
        );
}
```

## should_panic 이용하기

`should_panic`속성은 `#[test]`아래에 위치해야 하며 패닉이 일어나면 성공으로 간주한다.

```rust
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100,
got {}.", value);
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

- 위의 함수에서 `greater_than_100()` 함수는 panic이 발생하므로 성공한 테스트로 간주한다.
  아래처럼 서술하면 원하는 panic의 메세지를 체크할 수 있다.

```
mod tests {
        use super::*;
        #[test]
        #[should_panic(expected = "less than or equal to
100)]
        fn greater_than_100() {
                Guess::new(200);
        }
}
```

- 여기서는 원하는 메세지가 아니라면 테스트에 실패한 것으로 간주한다.

## 11.2 테스트 실행 방법 제어하기

- 테스트를 실행할 때는 기본적으로 스레드를 사용해 병렬 실행됨.

### 테스트시 쓰레드의 개수 지정하기

- `cargo test -- --test-threads=1`
- 이렇게하면 thread의 개수를 1개로 지정하며 여러개로 지정하여 병렬실행도 가능하다.

### 함수 출력 표시하기

- 테스트가 성공하면 출력 결과가 나오지 않게 된다.
- 테스트가 실패하게 되면 표준 출력으로 출력한 것들이 에러 메세지와 함께 출력된다.
- 성공한 테스트의 출력도 보고 싶다면 다음의 옵션을 사용하면 된다.
- `cargo test -- -- show-output`

### 선택해 테스트하기

- `cargo test test_function_name`
- 테스트 이름을 모두 입력하지 않고 일부만 입력하면 해당 테스트이름으로 시작하는 모든 테스트를 실행함.

### 테스트 함수 무시하기

- `#[ignore]` 속성을 가진 함수들은 특별한 호출이 있어야 실행이 된다.

```
#[test]
#[ignore]
fn expensive_test(){

}
```

- `cargo test -- --ignored`
  - 이 코드를 실행하면 무시된 함수들만 실행함.

## 11.3 테스트 조직화

### 유닛 테스트

유닛 테스트의 목적은 각 코드 단위를 나머지 코드와 분리해 제대로 작동하지 않는 코드가 어느 부분인지 빠르게 파악하는 것이다.
유닛 테스틑 src 디렉토리 내의 각 파일에 테스트 대상이 될 코드와 함께 작성한다.
각 파일에 tests 모듈을 만들고 `cfg(test)`를 어노테이션하는게 일반적인 관례다.

- `cfg` 속성은 설정을 의미함.
  - 이 아이템을 특정 설정 옵션 적용시에만 포함함.

### 테스트 모듈과 `#[cfg(test)]`

테스트 모듈에 어노테이션 해주는 `#[cfg(test)]`가 있으면 `cargo build`할 때 비들되지 않고 `cargo test` 명령어 실행 시에만 커파일 및 실행될 것을 러스트에게 전달함.

- 만약 통합 테스트 폴더가 있다면 해당 폴더에서는 어노테이션을 하지 않아주어도 됨.
- 단지 일반 파일과 분리하기 위한 코드임.

### 비공개 함수 테스트하기

러스트에서는 비공개 함수들 또한 테스트가 가능함.

```rust
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

- 이 코드에서 `internal_adder`함수는 `pub` 옵션이 달려있지 않지만,`super::*`를 use함으로써 비공개 함수를 호출할 수 있음.

## 통합 테스트

통합 테스트는 라이브러리와 완전히 분리된 환경에서 실행됨.

- 통합 테스트는 외부 코드와 마찬가지로 라이브러리의 공개 API만 호출 가능함.

### tests 디렉토리

- 통합 테스트를 작성하기 위한 디렉토리
- src폴더의 위치와 같은 공간에 작성함.