세상에서 제일 쉬운 러스트 프로그래밍

3장. 함수와 매크로

윤인도

freedomzero91@gmail.com

함수 선언

파이썬

```
def add(num1: int, num2: int) -> int:
    return num1 + num2
```

러스트

함수의 선언에 `fn` 키워드를 사용하고, 함수에서 실행할 코드를 중괄호로 묶어줍니다. 그리고 파이썬과 비슷하게 파라미터에는 `:i32`로 타입을 표기하고, 리턴값에는 `-> i32`처럼 화살표를 사용해 타입을 명시했습니다.

```
fn add(num1: i32, num2: i32) -> i32 {
    return num1 + num2;
}
```

> 이때 파이썬에서는 타입을 생략할 수 있지만, 러스트에서는 반드시 파라미터와 리턴 타입을 명시해야 한다는 점을 염두에 둬야 합니다. 타입이 잘못되거나 표기되지 않았다면 컴파일되지 않습니다.

러스트는 코드 마지막에서 `return` 키워드를 생략할 수 있습니다. 이때 세미콜론이 없다는 점에 주의하세요.

```
fn add(num1: i32, num2: i32) -> i32 {
    num1 + num2
}
```

이제 `add` 함수를 메인 함수에서 호출하고 값을 프린트해 보겠습니다.

```
fn add(num1: i32, num2: i32) -> i32 {
    num1 + num2
}

fn main() {
    println!("{}", add(1, 2));
}
```

실행 결과

```
3
```

키워드 파라미터

파이썬에서는 키워드 파라미터에 기본값을 사용할 수 있습니다.

```python
def add(num1: int, num2: int=1) -> int:
    return num1 + num2
```

러스트에서는 기본값을 설정하면 컴파일 에러가 발생

```rust
fn add(num1: i32, num2: i32=1) -> i32 {
    num1 + num2
}
```

여러 개의 값 리턴하기

파이썬에서 `swap` 이라는 함수를 아래와 같이 여러 개의 값을 튜플로 묶어서 리턴하도록 구현합니다.

```python
def swap(num1: int, num2: int) -> tuple[int, int]:
    return num2, num1


num1, num2 = swap(1, 2)
print(f"{num1}, {num2}")
```

실행 결과

```
2, 1
```

러스트도 여러 개의 값을 리턴하는 경우, 값들이 튜플로 묶이게 됩니다.
따라서 함수의 리턴 타입도 튜플로 `(i32, i32)` 표기합니다.

```
fn swap(num1: i32, num2: i32) -> (i32, i32) {
    (num2, num1)
}

fn main() {
    let (num1, num2) = swap(1, 2);
    println!("{num1}, {num2}");
}
```

실행 결과

```
2, 1
```

만일 함수에서 리턴하는 값이 없는 경우 리턴 타입을 생략하거나 `-> ()` 로 표기

```
fn print_hello() -> () {
    println!("Hello");
}
```

스코프

스코프(scope)란 변수에 접근할 수 있는 범위를 의미합니다. 먼저 파이썬에서는 스코프를 기본적으로 함수 단위로 구분합니다.

> 실제로는 파이썬은 LEGB 룰이라고 불리는 좀더 복잡한 스코프 규칙을 가지고 있지만, 여기서는 단순화해서 함수 기준으로 설명합니다.

```python
def hello(name: str):
    num = 3
    print(f"Hello {name}")


if __name__ == '__main__':
    my_name = "buzzi"

    if True:
        print("My name is", my_name)
        my_name = "mellon"

    hello(my_name)

    # print(num) # error
```

실행 결과

```
My name is buzzi
Hello mellon
```

러스트에서는 스코프를 `{}` 기준으로 구분

```
fn hello(name: String) {
    let num = 3;
    println!("Hello {}", name);
}

fn main() {
    let my_name = "buzzi".to_string();

    {
        println!("My name is {}", my_name);
        let my_name = "mellon";
    }

    hello(my_name);
}
```

실행 결과

```
My name is buzzi
Hello buzzi
```

`println!("{}", num); // error`를 마지막 줄에 추가하면,

```
Compiling rust_part v0.1.0 (/code/temp/rust_part)
error[E0425]: cannot find value `num` in this scope
--> src/main.rs:15:20
   |
15 |     println!("{}", num);
   |                    ^^^ not found in this scope
For more information about this error, try `rustc --explain E0425`.
error: could not compile `rust_part` due to previous error
```

익명 함수

익명 함수란 이름이 없는 함수라는 뜻으로, 프로그램 내에서 변수에 할당하거나 다른 함수에 파라미터로 전달되는 함수입니다. 따라서 익명 함수를 먼저 만들어 놓고 나중에 함수를 실행할 수 있습니다.

파이썬에서는 `lambda` 키워드를 사용합니다.

```
my_func = lambda x: x + 1
print(my_func(3))
```

러스트에도 람다 함수와 비슷한 개념이 있는데 바로 클로저(Closure)입니다.
클로저는 파라미터를 `| |` 의 사이에 선언하고, 그 뒤에 함수에서 리턴하는 부분을 작성합니다.

```
fn main() {
    let my_func = |x| x + 1;
    println!("{}", my_func(3));
}
```

이때 컴파일러가 클로저의 파라미터와 리턴값의 타입을 `i32`로 추측해서 보여줍니다.
하지만 타입을 명시하는 것도 가능합니다.

```
fn main() {
    let my_func = |x: i32| -> i32 { x + 1 };
    println!("{}", my_func(3));
}
```

파이썬에서 내부 함수(클로저)로 피보나치 수를 계산하는 예제는 다음과 같습니다.

```
def fibonacci(n):
    cache = {}

    def fib(n):
        if n in cache:
            return cache[n]
        if n < 2:
            return n
        cache[n] = fib(n - 1) + fib(n - 2)
        return cache[n]

    return fib(n)


fibonacci(10)
```

같은 로직을 러스트로 구현하면 클로저가 자기 자신을 부를 수 없기 때문에 컴파일되지 않습니다.

```rust
fn fib(n: u32) -> u32 {
    let cache = vec![0, 1];
    let _fib = |n| {
        if n < cache.len() {
            cache[n]
        } else {
            // _fib이 정의되기 전에 호출
            let result = _fib(n - 1) + _fib(n - 2);
            cache.push(result);
            result
        }
    };

    _fib(n)
}
fn main() {
    println!("{}", fib(10));
}
```

매크로

매크로는 **다른 코드를 생성하는 코드**를 작성하는 방법으로,
메타 프로그래밍(meta programming)이라고도 합니다.

> 매크로 문법은 상당히 복잡하기 때문에 자세한 문법은 여기서 설명하지 않습니다.

러스트의 함수는 가변 길이의 파라미터를 만들 수 없음!

→ 매크로를 사용!

예:

- `println!("hello")`와 같이 하나의 인수로 `println!`을 호출하거나
- `println!("hello {}", name)`과 같이 두 개의 인수로 호출할 수 있습니다.

파이썬에서는 `*args`를 사용해 가변 길이 파라미터를 사용

```python
def get_sum(*args):
    return sum(args)

print(get_sum(1, 2)) # 3
print(get_sum(1, 2, 3))  # 6
```

러스트

```rust
macro_rules! get_sum {
    // 쉼표로 구분된 임의의 개수의 식을 입력으로 받습니다.
    ($($x:expr),*) => {{
        // 식들을 벡터에 담습니다.
        let args = vec![$($x),*];

        // 벡터를 반복하며 요소들의 합을 구합니다.
        args.iter().sum::<i32>()
    }};
}

fn main() {
    println!("{}", get_sum!(1, 2)); // 3
    println!("{}", get_sum!(1, 2, 3)); // 6
}
```

대부분의 상황에서 매크로를 직접 작성해야 하는 경우는 별로 없을 것입니다.
매크로에 대해서 더 자세히 알고 싶다면 러스트 공식 문서를 참고하세요.

연습문제

1. 두 개의 정수를 인자로 받아 두 정수의 곱을 반환하는 함수를 작성해 보세요.

```
fn multiply_numbers(?) {?}


fn main() {
    let result = multiply_numbers(3, 4);
    println!("The product of 3 and 4 is: {}", result); // 12
}
```

정답

```rust
fn multiply_numbers(a: i32, b: i32) -> i32 {
    a * b
}

fn main() {
    let result = multiply_numbers(3, 4);
    println!("The product of 3 and 4 is: {}", result);
}
```

2. 두 개의 정수를 인자로 받아 두 정수의 곱을 반환하는 클로저를 작성해 보세요.

```
fn main() {
    let multiply_numbers = |?| -> ? {  };

    let result = multiply_numbers(3, 4);
    println!("The product of 3 and 4 is: {}", result); // 12
}
```

정답

```rust
fn main() {
    let multiply_numbers = |a: i32, b: i32| -> i32 { a * b };

    let result = multiply_numbers(3, 4);
    println!("The product of 3 and 4 is: {}", result);
}
```