2_Functions_en

May 2, 2023

1 Functions

Function definitions in Rust start with fn and have a set of parentheses after the function name. The curly brackets tell the compiler where the function body begins and ends.

Rust code uses snake case as the conventional style for function and variable names. In snake case, all letters are lowercase and underscores separate words. Here's a program that contains an example function definition:

[ ]:
```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
main();
```

We can call any function we've defined by entering its name followed by a set of parentheses. Note that we defined `another_function` after the `main` function in the source code; we could have defined it before as well. Rust doesn't care where you define your functions, only that they're defined somewhere.

2 Function Parameters

Functions can also be defined to have parameters. In function signatures, you must declare the type of each parameter. This is a deliberate decision in Rust's design: requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what you mean.

[ ]:
```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
```

```
}
main();
```

[ ]:
```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
main();
```

3 Statements and Expressions

Function bodies are made up of a series of statements optionally ending in an expression. **Statements** are instructions that perform some action and *do not return a value*. **Expressions** evaluate to a resulting value.

[ ]:
```
fn main() {    // the entire function definition is a statement
    let y = 6; // this is a statement
}
main();
```

3.1 Statements

Statements do not return values. Therefore, you can't assign a let statement to another variable. This is different from what happens in other languages, such as C and Ruby, where the assignment returns the value of the assignment. Wrong assignement example:

[ ]:
```
fn main() {
    let x = (let y = 6); // error
}
main();
```

3.2 Expressions

Expressions evaluate to something. Examples: math operations, a function call, a macro call. The block that we use to create new scopes, **{}**, is an expression:

[ ]:
```
fn main() {
    let x = 5;

    let y = {  // this block is an expression that evaluates to 4
        let x = 3;
        x + 1 // the last expression is the value for the block
    };
```

```
    println!("The value of y is: {}", y);
}
main();
```

Note the `x + 1` line without a semicolon at the end, which is unlike most of the lines you've seen so far. Expressions do not include ending semicolons. If you add a semicolon to the end of an expression, you turn it into a statement, which will then not return a value.

4 Functions with Return Values

Functions can return values to the code that calls them. We don't name return values, but we do declare their type after an arrow (`->`). In Rust, the return value of the function is synonymous with the value of the final expression in the block of the body of a function. You can return early from a function by using the return keyword and specifying a value, but most functions return the last expression implicitly.

[ ]:
```
fn five() -> i32 {  // function has a return type of i32
    5    // this is the return expression (no semicolon)
}

fn main() {
    let x = five();   // this code is equivalent to: let x = 5;

    println!("The value of x is: {}", x);
}
main();
```

[ ]:
```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;  // a semicolon here results in an error
}
main();
```

5 Exercises

Exercise 1: Write a function that determines whether 2 values are equals with a tolerance of `EPSILON`.

Sample test data:

- (0.00009384921 , 0.00009384922)
- (0.000093849221 , 0.000093849222)

[ ]:
```rust
/// The standard value for epsilon 1e-11
const EPSILON: f64 = 1e-11;

/// TO DO (1) Implement a function called eq

fn main() {
    let x = 0.000093849221;
    let y = 0.000093849222;
    println!("The numbers are equal: {}", eq(x,y));
}
main();
```

Solution:

[ ]:
```rust
/// The standard value for epsilon 1e-11
const EPSILON: f64 = 1e-11;

/// Determines whether x and y are equals with a tolerance of eps
fn eq(x: f64, y: f64) -> bool {
    (x - y).abs() < EPSILON
}

fn main() {
    let x = 0.000093849221;
    let y = 0.000093849222;
    println!("The numbers are equal: {}", eq(x,y));
}

main();
```