Programming 101: Introduction to Programming Concepts with Rust

Introduction

Welcome to **Programming 101**! This guide introduces basic programming concepts using the Rust language. Whether you're new to programming or just learning Rust, this document will help you understand fundamental ideas and provide code examples to build your skills.

Basic Concepts

Variables and Data Types

Variables store values that can be used in a program. In Rust, variables are declared using the `let` keyword. Rust supports various data types like integers, floating-point numbers, booleans, and strings.

```rust
fn main() {
    // Immutable variable
    let x = 5; // i32 (integer)

    // Mutable variable
    let mut y = 10; // i32 (integer)
    y = 15; // Now y is 15

    // Float type
    let pi: f64 = 3.14;

    // Boolean type
    let is_rust_fun = true;

    // String type
    let greeting = String::from("Hello, Rust!");

    println!("x: {}, y: {}, pi: {}, is_rust_fun: {}, greeting: {}", x, y, pi, is_rust_fun, greeting);
}
```

Listing 1: Variable declaration and data types in Rust

Control Flow

Control flow allows you to make decisions in your code using `if`, `else`, and loops like `while` and `for`.

```rust
        fn main() {
           let age = 18;

           // If-Else statement
           if age >= 18 {
              println!("You are an adult.");
           } else {
              println!("You are a minor.");
           }

           // For loop
           for i in 0..5 {
              println!("Loop iteration: {}", i);
           }

           // While loop
           let mut count = 0;
           while count < 3 {
              println!("Count: {}", count);
              count += 1;
           }
        }
```

Listing 2: If statements and loops

Functions

Functions are reusable blocks of code that perform a specific task. They are declared using the `fn` keyword in Rust.

```rust
        fn main() {
           greet("Rust");
           let sum = add(5, 10);
           println!("Sum: {}", sum);
        }

        // Function to print a greeting
        fn greet(name: &str) {
           println!("Hello, {}!", name);
        }

        // Function to add two numbers
        fn add(a: i32, b: i32) -> i32 {
           a + b // Implicit return
        }
```

Listing 3: Functions in Rust

Advanced Concepts in Rust

Rust introduces unique features such as **ownership**, **borrowing**, and **lifetimes**, which help ensure memory safety.

Ownership

In Rust, each value has a single owner. When ownership is transferred, the original owner can no longer use the value.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // Ownership is moved to s2

    // println!("{}", s1); // Error: s1 is no longer valid
    println!("{}", s2); // s2 is valid
}
```

Listing 4: Ownership in Rust

Borrowing

Borrowing allows a value to be accessed without transferring ownership, using references.

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // Borrow s1
    println!("Length of '{}': {}", s1, len); // s1 is still valid
}

fn calculate_length(s: &String) -> usize {
    s.len() // Use the reference
}
```

Listing 5: Borrowing in Rust

Lifetimes

Lifetimes ensure that references are valid as long as needed. They prevent dangling references.

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("Rust");
    let string2 = String::from("Programming");
    let result = longest(&string1, &string2);
    println!("The longest string is '{}'", result);
}
```

Listing 6: Lifetimes in Rust

Conclusion

This guide has introduced basic programming concepts, control flow, and Rust-specific features like ownership and borrowing. These foundations will help you as you start coding the virtual robot to navigate the maze!