The Introduction to Rust

Hands-on exercises

Please note that for each of the topics covered here, a corresponding exercise (and more) can be found at https://github.com/rust-lang/rustlings/tree/main/exercises. Readers are strongly encouraged to refer to the exercises for each topic.

Getting started with Rust

Reference: https://doc.rust-lang.org/book/ch01-00-getting-started.html

Installing and getting started with a ‘Hello, world!’ in Rust. Please note that for newcomers, *unix-based systems are the easiest to work with Rust. These commands assume a *unix-based system. The commands for Windows may vary, please refer to the book mentioned above.

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

```
fn main() {
    println!("Hello, world!");
}
```

```
rustc main.rs
```

```
./main
```

Cargo is Rust’s package manager, somewhat similar to npm in JavaScript but more. It should come along with the Rust installation.

```
cargo --version
```

Check out more on the basics of Cargo at: https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Basic data types in Rust

Reference: https://doc.rust-lang.org/book/ch03-02-data-types.html

As Rust is a strongly-typed language, it is important to understand how type definitions work in Rust. Please go through the reference to see the details of basic Rust types.

Functions in Rust

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

    second_function();
}

fn second_function() {
    println!("A second function.");
}
```

We define a function in Rust by entering fn followed by a function name and a set of parentheses. The curly brackets tell the compiler where the function body begins and ends.

Learn more about functions at: https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

Flow Control Statements

```
fn main() {
    let number = 60;

    if number % 7 == 0 {
        println!("number is divisible by 7");
    } else if number % 5 == 0 {
        println!("number is divisible by 5");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 7, 5, or 2");
    }
}
```

This program has four possible paths it can take, based on the conditions specified for each block. Can you guess what will be the output of this program?

To learn more about control statements, read: https://doc.rust-lang.org/book/ch03-05-control-flow.html

Loops in Rust

```
fn main() {
    let mut i:i32 = 1;
    let result = loop{
        if i==10 {
            break i;
        }
        i+=1;
    };
    println!("The value of i is now {}",result);
}
```

The simplest type of loop here is using the `loop` keyword, which also allows you to return values from the loop body, as shown in the above snippet.

```
fn main() {
    let mut i:i32 = 1;
    while i<=10 {
        println!("The value of i is {}",i);
        i+=1;
    }
}
```

Another way of looping through statements is with a `while` loop.

You can learn more about loops at: https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

Enum and Match Statements

Enums are a way of defining custom data types in a different way than you do with structs.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
```

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

You can learn more on enums at: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

To read about the match statement, visit: https://doc.rust-lang.org/book/ch06-02-match.html

Collections: Vectors and Hash Maps

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory.

```
let v: Vec<i32> = Vec::new();
v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

HashMap stores a mapping of keys of type K to values of type V. It does this via a hashing function, which determines how it places these keys and values into memory.

```
fn main() {
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
}
```

To read more on collections, visit: https://doc.rust-lang.org/book/ch08-00-common-collections.html

Macros

Fundamentally, macros are a way of writing code that writes other code, which is known as metaprogramming.

Metaprogramming is useful for reducing the amount of code you have to write and maintain, which is also one of the roles of functions. However, macros have some additional powers that functions don't.

A function signature must declare the number and type of parameters the function has. Macros, on the other hand, can take a variable number of parameters.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Learn more about macros at: https://doc.rust-lang.org/book/ch19-06-macros.html

Generics

We can use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types.

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Read more on: https://doc.rust-lang.org/book/ch10-01-syntax.html

Traits

A trait tells the Rust compiler about functionality a particular type has and can share with other types. We can use traits to define shared behaviour in an abstract way. We can use trait bounds to specify that a generic type can be any type that has certain behaviour.

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Learn more on: https://doc.rust-lang.org/book/ch10-02-traits.html