🦀 Rust 🦀

What is Rust?

Rust is commonly described as

> A memory safe, compiled language, with high level simplicity and low level performance.

Lets go through each of these one by one, and discuss what it means. I will be using a total of 3 other languages: Python, Java, and C, to try and showcase the features in Rust. This is done in a hope that you are at least familiar with at least one of them and that will help you draw connections to what I am describing.

- **compiled**: this one is very simple, and it just means that before you can execute you code, you need to first compile it into an intermediate file. This is in contrast to JIT (just in time) languages like python, which execute the file directly. This is a great thing for performance because the file can be extensively optimized by the compiler, improving runtime speed by several magnitudes. Rust is compiled using the **rustc** tool.
- **high level simplicity & low level simplicity**: also known as zero-cost abstraction, is a design paradigm in which you have a systems language that implements many features you would find in modern day scripting languages. This is done by implementing these features in a way that does not add any overhead to the program (zero-cost abstraction). This is accomplished by using a powerful compiler to optimize the code, and by using a garbage collection system such as ownership-borrow.
- **memory safe**: memory safety might not be a term you are familiar with if you have not used a low level language like 'C' before. Most higher level languages out there (Python, Java, etc.), implement a a garbage collector (GC), which does memory management for you. Whenever you create a variable in any language, and a assign it a value, that variable needs to be stored in memory. When that variable goes out of scope, aka it is not used anymore, then that memory needs to be freed at some point. If this does not happen, your entire computer can eventually freeze because there is no more memory for it to use. Memory is freed in three ways: through a GC, manually, or an ownership-borrow system.
    - **GC**: a garbage collector is a periodic process that your program executes in order to free up memory. It works by looping over all the memory in the current scope and checking if there are any references left to that memory in your program. If there aren't any, then that memory can be freed, which just means it is marked as no longer used, and returned to the Operating System to manage. Unfortunately this can create big lag spikes in your program whenever the GC executes, since checking all the memory in your program can take quite a bit of time depending on how large your program gets.
    - **manual**: manual memory management is where you explicitly request memory, most commonly through the malloc() function, and free memory, most commonly through the free() function.
    - **ownership-borrow**: the ownership-borrow system is a form of memory management in which the compiler always knows who owns what part of memory because each part of memory only has one owner. This is incredibly efficient because this means that if the owner goes out of

scope, that part of memory can be freed immediately, without any concerns of dangling pointers and NullPointerExceptions, that come with manual memory management.

<Rust Syntax>

> Rust syntax is familiar, straight forward, and easy to understand.

- Variables can be declared with the *let* keyword, followed by its name and value.

```
let x = 5;
```

Variables can optionally be typed using the following syntax

```
let x: i32 = 5;
```

However this is not strictly necessary for variable declarations since all variables have a default type, in the case of an integer its an i32, aka a signed 32 bit integer.

We can specify that we only want a signed 8 bit integer like this:

```
let x: i8 = 5;
```

Or even that we want an unsigned 16 bit integer like this:

```
let x: u16 = 5;
```

The full list of Rust's types can be found here

> https://doc.rust-lang.org/reference/types.html

- Functions are next and are declared by the *fn* keyword, followed by the function name, comma separated parameters (all of which must be typed), and the return type (which can only be omitted if there is none).

```
fn add_two_nums(a: i32, b: i32) -> i32 {
    let sum: i32 = a + b;
    return sum;
}
```

Let's change this function very slightly to add the first number to the second, and print the result:

```
```

```
fn add_a_to_b(a: i32, b: i32) {
    a = a + b;
    println!("{a}");
}
```

If we try executing this we get this error:

```
error[E0384]: cannot assign to immutable argument `a`
--> main.rs:7:2
|
6 | fn add_a_to_b(a: i32, b: i32) -> i32 {
|               - help: consider making this binding mutable: `mut a`
7 |     a = a + b;
|     ^^^^^^^^^ cannot assign to immutable argument
```

This is happening because of the fact that all variables in rust are immutable by default. Once you assign a variable a value, you cannot reassign it. We can solve this by specifying that the a parameter is mutable using the *mut* keyword:

```
fn add_a_to_b(mut a: i32, b: i32) {
    a = a + b
    println!("{a}");
}
```

This function compiles successfully.

Demo

Lets write a simple function that takes in two bounds, and sums all the even numbers between them. We will call this function *sum_even_numbers*. I will first write out this function in Python, and then translate it to Java, then C, and finally to Rust. I will write this as explicitly as possible to avoid any confusion. We will simplify once we get to the Rust version, and this will also give you the chance to learn more about Rust syntax.

- ```
  def sum_evens(a: int, b: int):
      if (a <= b):
          sum: int = 0
          for i in range(a, b+1):
              if i % 2 == 0:
                  sum += i
          return sum
      else:
          return 0
  ```
  ```

Take your time to read through this function, and make sure you understand what each part is doing.

- ```java
  public static int sum_evens(int a, int b) {
      if (a <= b) {
          int sum = 0;
          for (int i = a; i <= b; i++) {
              if (i % 2 == 0) {
                  sum += i;
              }
          }
          return sum;
      }
      else {
          return 0;
      }
  }
  ```

Apologies for any PTSD this may have caused you from your APCS classes, but as you can see it is very similar to the Python version, apart from the for loop.

- ```c
  int sum_evens(int a, int b) {
      if (a <= b) {
          int sum = 0;
          for (int i = a; i <= b; i++) {
              if (i % 2 == 0) {
                  sum += i;
              }
          }
          return sum;
      }
      else {
          return 0;
      }
  }
  ```

The C version of this function is almost identical to the Java version, so there is not much to talk about here. But do try and notice the subtle differences between the styles of the two languages.

Now we are going to move on to the Rust version. But because we have an simplified version, we will create multiple versions of this function, each one more simplified than the last.

- ```rust
  fn sum_evens1(a: i32, b: i32) -> i32 {
      if a <= b {
          let mut sum: i32 = 0;
          for i in a..=b { // What is a..=b?
              if i % 2 == 0 {
                  sum += i;
              }
  ```
  ```

```
        }
        return sum;
    }
    else {
        return 0;
    }
}
```

This is the first version of our function, and it is very similar to the Python version. But there are a few things that are different. First of all, the for loop is different. In Python, we used the range() function to create a range of numbers, and then looped over that range. In Java, we used a for loop to loop over a range of numbers. In Rust, we use the ..= operator to create a range of numbers, and then loop over that range. This is a very common pattern in Rust, and is called an iterator.

- ```
  fn sum_evens2(a: i32, b: i32) -> i32 {
      // a..=b is a range, from a to b inclusive.
      // if a > b, the range is empty, therefore sum is 0.
      let mut sum = 0;
      for i in a..=b {
          if i % 2 == 0 {
              sum += i;
          }
      }
      return sum;
  }
  ```

In this second version of our function we are simplifying it down some more by removing the unnecessary checks and more explicit if else statements.

- ```
  fn sum_evens3(a: i32, b: i32) -> i32 {
      // i % 2 == 0 is a filter, lets use the filter method
      let mut sum = 0;
      for i in (a..=b).filter(|e| e % 2 == 0) {
          sum += i;
      }
      return sum;
  }
  ```

In this third version of our function we are using the filter method to filter out all the odd numbers. This is a very common pattern in Rust, and is called a pipeline. Pipelines create more declarative code, and make it easier to read and understand since its reads more like a sentence rather than a mathematical equation or logical statement.

- ```
  fn sum_evens4(a: i32, b: i32) -> i32 {
      // sum += i is a summation, lets use the sum method
      let sum = (a..b+1).filter(|e| e % 2 == 0).sum();
  ```
  ```

```
    return sum;
}
```

We take this one step further by using the sum method to sum all the numbers in the range.

- ```
  fn sum_evens5(a: i32, b: i32) -> i32 {
      // more straight forward
      return (a..b+1).filter(|e| e % 2 == 0).sum();
  }
  ```

We could leave the function here as it is, but if you want to be more "Rusty" then we can adapt it even more. But from here on out, the changes are more based on personal preference, and not necessarily better or worse in terms of readability.

- ```
  fn sum_evens6(a: i32, b: i32) -> i32 {
      // using "correct" rust syntax:
      // 1. no need for return, rust returns last line of function by
  default (must omit semicolon at end of line)
      // 2. chain methods on new lines
      (a..b+1)
          .filter(|e| e % 2 == 0)
          .sum()
  }
  ```

We have finally reached the end of our journey. We have taken a simple function, that took you a little bit to read and understand, and converted it into the same function, in terms of functionality and performance, but in a much more readable and understandable way.