Variables, Functions and If Else, For loop

```
fn main() {

    //Handling the variables.
    let x: u8 = 7; //defining x with data type u8. But we don't need to define it because Rust can
automatically assess the data type.
    println!("the value of x is {}", x);

    let y = &x+1; //here we are borrowing (pointing to x) instead of actually owning the variable x.
    //within a scope, a variable can either be borrowed or can be owned by one owner at any
given point of time.
    println!("the value of y is {}", y);

    //mutable variables, by default the variables are immutable, if you wish to change the
variable, then
    //it should be declared as mutable.

    let mut z = 5;
    println!("The value of z is {}", z);
    z = 7;
    println!("The amended value of z is {}", z);

    //shadowning the variable - You must use let - usually we cannot assign a variable twice, but
this is shadowing.

    let x = x+1; //this is called as shadowning immutable variable. Here we are updating x without
it being mutable.
    println!("the value of x after shadowing is {}", x);



    //Constants must be in all caps with underscore in between. They can't be mutated.

    const SECONDS_IN_HOUR : u16 = 60*60*1;
    println!("total seconds in an hour are {}", SECONDS_IN_HOUR);

    let p = add_one(x);
    println!("Value of p {}", p);

    loop_print(p);
}
```

```
//Functions are defined with parameters and return data type. A function can have multiple parameters
fn add_one(number: u8) -> u8 {

    number + 1

}

//in this case the function will not return anything, but just print the values.
fn loop_print(number: u8) {

    for i in 1..number {

        if i==5 { println!("number is 5 now")};
        println!("{}", i);
    }

}
```

—---

Struct, Method and Debug

```
#[derive(Debug)] //Derive debug is used in dev mode to print the background functioning of the code.
//We can use dbg! macro to print any line that we wish to inpect - "What's going on behind"

// Struct is like a document of MongoDB where we can define objects with different/same data-types.
struct Rectangle{
    width: u8,
    height: u8
}

//This is a method syntax. Methods are basically functions but they have their own Struct, Enums.
//Methods will have a same name as Struct. Methods can have multiple functions inside it, each function can have multiple parameters.


impl Rectangle {
    fn area(&self) -> u8 {
            self.width * self.height
    }

    fn twomethods(&self, second: Rectangle) -> bool {
        (self.width*self.height > 0) && (second.width* second.height>0)

    }
}


fn main() {

//A variable can have a data type as Struct. and this is how we define it.

    let first = Rectangle {
        width: 8,
        height: 9
    };

    let second = Rectangle {
        width: 10,
        height: 12
    };

dbg!(&first); //dbg! is used to print any expression or element that can't be printed using print macro.

//println!("Area is {}", first.area()); //the way we call method is variable.function inside method.
println!("The area of the rectangle is {}", first.area());
println!("The result of comparison is {}", first.twomethods(second));

}
```

Airtable Get request:

```
use reqwest::Client;
use std::error::Error;
use std::env;
use dotenv::dotenv;


#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {

    dotenv().ok();

    // Read the Airtable API Key and Base ID from environment variables
    let api_key = env::var("AIRTABLE_API_KEY")?;
    let base_id = env::var("AIRTABLE_BASE_ID")?;
    let table_name = env::var("AIRTABLE_TABLE_ID")?;  // Replace with your table name

    let query_params = vec![
        ("fields[]", "fldHlPwYeLXksjSpU"), //Airtable API takes in Fields as an array in query and hence fields[]
        ("fields[]", "fldsc4SUcyC8459GQ"),
        ("fields[]", "fldu7bBf7MKQqlzLo"),
        //("fields[]", "fldON2AoXyUWfx1Pf"),
        ("filterByFormula", "fldsc4SUcyC8459GQ = 'UK'"),
        //("returnFieldsByFieldId", "true"),
    ];

    // Build the Airtable API URL
    let url = format!("https://api.airtable.com/v0/{}/{}", base_id, table_name);

    // Create an HTTP client
    let client = Client::new();

    // Send the GET request
    let response = client
        .get(&url)
        .header("Authorization", format!("Bearer {}", api_key))
        .query(&query_params)
        .send()
        .await?;

    // Check if the request was successful
    if response.status().is_success() {
        // Parse and print the response body
        let response_text = response.text().await?;
        println!("Response: {}", response_text);
    } else {
        // Print error status
        println!("Error: {:?}", response.status());
    }

    Ok(())
}
```