Cheat Sheet - Rust (Basics for Python)

esuriddick

Documentation

Non-Docoumentational Comment

```
// A single or multiple line of comments can
// be done like this

/* Alternatively, to avoid repeating the comment
marker, you can nest it like this */
```

Docoumentational Comment

```
/// Supports markdown notation.
/// # Examples
///
/// “‘
/// let five = 5
///
```

Bindings

Structures

```
#[derive(Debug)]
struct Point {  // Define elements and data type
        x: i32,
        y: i32,
}
fn main() {
    let origin: Point = Point { x: 0, y: 0 };
    println!("Structure: {:?}", origin);
}
```

Enumerations

```
#[derive(Debug)]
enum Direction {
        Left,
        Right,
        Up,
        Down,
}
fn main() {
    let up = Direction::Up;
    println!("Direction: {:?}", up)
}
```

Numbers

```
fn main() {
    let x: i32 = 1;
    let mut y: f64 = 10.0;
    y += 5.0;
    println!("Value of x: {}, and of y: {}", x, y);
}
```

Strings

```
fn main() {
    let x: &str = "Hello, world";
    let y: String = "Hello, world".to_string();
    let z: &str = &x;
    println!("Literal string: {}", x);
    println!("Non-literal string: {}", y);
    println!("View/slice of x: {}", z);
}
```

Vectors / Arrays

```
fn main() {
    // Fixed-size array
    let four_ints: [i32; 4] = [1, 2, 3, 4];

    // Dynamic-sized array
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    vector.push(5); // Append element(s) to vector
    let slice: &[i32] = &vector;

    println!("Fixed-size array: {:?}", four_ints);
    println!("Dynamic-sized array: {:?}", vector);
    println!("View/slice of vector: {:?}", slice);
}
```

Tuples

```
fn main() {
    let x: (i32, &str, f64) = (1, "hello", 3.4);
    let (a, b, c) = x;
    println!("{} {} {}", a, x.1, c);
}
```

Functions

Function

```
fn main() {
    fn fibonacci(n: i32) -> i32 {
        match n {
            0 => 1,
            1 => 1,
            _ => fibonacci(n - 1) + fibonacci(n - 2),
        }
    }
}
```

Function Pointer

```
fn main() {
    type FunctionPointer = fn(i32) -> i32;
    let fib : FunctionPointer = fibonacci;
    println!("Fibonacci sequence: {}", fib(4));
}
```

Control Flow

If Statement

```
fn main(){
    if 1 == 1 {
        println!("Maths is working!");
    } else if 5 > 2 {
        println!("What is going on?")
    } else {
        println!("Oh no...");
    }
}
```

If as an Expression

```
fn main(){
    let value: &str = if true {
        "good"
    } else {
        "bad"
    };
    println!("Value: {}", value);
}
```

Ranges

```
fn main(){
    for i in 1u32..4 {
        print!("Range {}\n", i);
    }
}
```

For Loop

```
fn main(){
    let array = [1, 2, 3];
    for i in array {
        println!("Loop {}", i);
    }
}
```

While Loop

```
fn main(){
    let mut counter: i32 = 1;
    while counter < 4 {
        println!("Counter: {}", counter);
        counter += 1;
    }
}
```

Error Handling

Result Type

Rust uses Result for functions that might succeed or fail.

```
fn divide(x: i32, y: i32) -> Result {
    if y == 0 {
        Err("Cannot divide by zero".to_string())
    } else {
        Ok(x / y)
    }
}

fn main() {
    match divide(10, 2) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }
}
```

Option Type

Rust uses Option for values that might be there or might be missing.

```
fn find_number(arr: &[i32], target: i32) -> Option {
    for &num in arr.iter() {
        if num == target {
            // Return the found number wrapped in Some
            return Some(num);
        }
    }
    // Return None if the number is not found
    None
}

fn main() {
    let numbers = [1, 2, 3, 4, 5];

    // Search for number that exists
    match find_number(&numbers, 3) {
        Some(num) => println!("Found the number: {}",
        num),
        None => println!("Number not found"),
    }

    // Search for number that doesn't exist
    match find_number(&numbers, 6) {
        Some(num) => println!("Found the number: {}",
        num),
        None => println!("Number not found"),
    }
}
```