Assignment # 2

***Async Rust***

What does ***async*** mean?

In Rust, when we talk about async, we’re talking about running code concurrently, or having multiple overlapping (in time) computations run on a single thread. Multithreading is a related, but distinct concept. Multithreading is ideal for when you’ve got computationally intensive tasks (so-called *CPU-bound* tasks) that can be spread across multiple, separated cores. Concurrent programming is better suited for when the task spends a lot of time waiting, such as for a response from a server. These tasks are called *IO-bound*.

So asynchronous programming lets us run multiple of these IO-bound computations at the same time on a single thread. They can run at the same time because when they’re waiting for a response, they’re just idle, so we can let the computer keep working on something that isn’t waiting. When we reach a point where we need the result of an asynchronous computation, we must .await it. In Rust, values that are ‘awaitable’ are known as ‘futures’.

An async function does not (necessarily) start executing immediately

To start an asynchronous function, you must either .await it or launch a task using an executor (we’ll get to that in a moment). Until this happens, all you have is a Future that has not started. Let’s look at an example to make it clearer:

use **async\_std::**task;

// ^ we need this for task spawning

async fn negate\_async(n: i32) -> i32 {

**println!**("Negating {}", n);

**task::**sleep(**std::time::Duration::**from\_secs(5)).await;

**println!**("Finished sleeping for {}!", n);

n \* -1

}

async fn f() -> i32 {

let neg = negate\_async(1);

// ... nothing happens yet

let neg\_task = **task::**spawn(negate\_async(2));

// ^ this task /is/ started

**task::**sleep(**std::time::Duration::**from\_secs(1)).await;

// we sleep for effect.

neg.await + neg\_task.await

// ^ this starts the first task `neg`

// and waits for both tasks to finish

}

You need an external library to use async/.await

As was briefly alluded to above, you need to reach for an external library to do asynchronous programming in Rust. This took me a while to understand, as I’m used to it being part of the language experience. In Rust, however, **you need a dedicated executor**[1](https://thomashartmann.dev/blog/async-rust/#fn1). The executor is what takes care of executing the futures, polling them and returning the results when they’re done. The standard library does not come with an executor, so we need to reach out to an external crate for this. There are a few ones to choose from, but the two most prominent ones are [async-std](https://async.rs/) (which we’re using here) and [tokio](https://tokio.rs/).

## A minimal async example!

# For preparation, make sure you’ve got at least version 1.39 of Rust and cargo available.

creating the application

Let’s create a new application! Simply run this command in your preferred directory:

cargo new async-basics

Dependencies

We’re going to be using [async-std](https://crates.io/crates/async-std) for spawning tasks, and [surf](https://crates.io/crates/surf) to fetch data from the API. Let’s add them to the Cargo.toml file. Your whole file should look something like this:

[package]

name = "async-basics"

version = "0.1.0"

authors = ["Your Name <your.email@provider.tld>"]

edition = "2018"

[dependencies]

async-std = "1"

surf = "1"

Fetch data

Okay, final step. Let’s modify the main.rs file. We’ll make it as simple as possible. Here’s what we want to use:

use **async\_std::**task;

use surf;

// fetch data from a url and return the results as a string.

// if an error occurs, return the error.

async fn fetch(url: &str) -> Result<String, **surf::**Exception> {

**surf::**get(url).recv\_string().await

}

// execute the fetch function and print the results

async fn execute() {

match fetch("https://pokeapi.co/api/v2/move/surf").await {

**Ok**(s) => **println!**("Fetched results: {:#?}", s),

**Err**(e) => **println!**("Got an error: {:?}", e),

};

}

fn main() {

**task::**block\_on(execute());

// ^ start the future and wait for it to finish

}