

[Rustlings](#) as an example of CodeSlide CLI

rustlings 🦀❤️

Greetings and welcome to `rustlings`. This project contains small exercises to get you used to reading and writing Rust code. This includes reading and responding to compiler messages!

...looking for the old, web-based version of *Rustlings*? Try [here](#)

Alternatively, for a first-time Rust learner, there are several other resources:

- [The Book](#) - The most comprehensive resource for learning Rust, but a bit theoretical sometimes. You will be using this along with Rustlings!
- [Rust By Example](#) - Learn Rust by solving little exercises! It's almost like `rustlings`, but online

Getting Started

Note: If you're on MacOS, make sure you've installed Xcode and its developer tools by typing `xcode-select --install`. Note: If you're on Linux, make sure you've installed gcc. Deb: `sudo apt install gcc`. Yum: `sudo yum -y install gcc`.

You will need to have Rust installed. You can get it by visiting <https://rustup.rs>. This'll also install Cargo, Rust's package/project manager.

MacOS/Linux

Just run:

```
curl -L https://raw.githubusercontent.com/rust-lang/rustlings/main/install.sh | bash
```

Or if you want it to be installed to a different path:

```
curl -L https://raw.githubusercontent.com/rust-lang/rustlings/main/install.sh | bash -s mypath/
```

This will install Rustlings and give you access to the `rustlings` command. Run it to get started!

Nix

Basically: Clone the repository at the latest tag, finally run `nix develop` or `nix-shell`.

```
# find out the latest version at https://github.com/rust-lang/rustlings/releases/latest (on edit 5.4.1)
git clone -b 5.4.1 --depth 1 https://github.com/rust-lang/rustlings
cd rustlings
# if nix version > 2.3
nix develop
# if nix version <= 2.3
nix-shell
```

Windows

In PowerShell (Run as Administrator), set `ExecutionPolicy` to `RemoteSigned`:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Then, you can run:

```
Start-BitsTransfer -Source https://raw.githubusercontent.com/rust-lang/rustlings/main/install.ps1 -Destination
$env:TMP/install_rustlings.ps1; Unblock-File $env:TMP/install_rustlings.ps1; Invoke-Expression
$env:TMP/install_rustlings.ps1
```

To install Rustlings. Same as on MacOS/Linux, you will have access to the `rustlings` command after it. Keep in mind that this works best in PowerShell, and any other terminals may give you errors.

If you get a permission denied message, you might have to exclude the directory where you cloned Rustlings in your antivirus.

Browser



Manually

Basically: Clone the repository at the latest tag, run `cargo install --path ..`

```
# find out the latest version at https://github.com/rust-lang/rustlings/releases/latest (on edit 5.4.1)
git clone -b 5.4.1 --depth 1 https://github.com/rust-lang/rustlings
cd rustlings
cargo install --force --path .
```

If there are installation errors, ensure that your toolchain is up to date. For the latest, run:

```
rustup update
```

Then, same as above, run `rustlings` to get started.

Doing exercises

The exercises are sorted by topic and can be found in the subdirectory `rustlings/exercises/<topic>`. For every topic there is an additional README file with some resources to get you started on the topic. We really recommend that you have a look at them before you start.

The task is simple. Most exercises contain an error that keeps them from compiling, and it's up to you to fix it! Some exercises are also run as tests, but `rustlings` handles them all the same. To run the exercises in the recommended order, execute:

```
rustlings watch
```

This will try to verify the completion of every exercise in a predetermined order (what we think is best for newcomers). It will also rerun automatically every time you change a file in the `exercises/` directory. If you want to only run it once, you can use:

```
rustlings verify
```

This will do the same as `watch`, but it'll quit after running.

In case you want to go by your own order, or want to only verify a single exercise, you can run:

```
rustlings run myExercise1
```

Or simply use the following command to run the next unsolved exercise in the course:

```
rustlings run next
```

In case you get stuck, you can run the following command to get a hint for your exercise:

```
rustlings hint myExercise1
```

You can also get the hint for the next unsolved exercise with the following command:

```
rustlings hint next
```

To check your progress, you can run the following command:

```
rustlings list
```

Testing yourself

After every couple of sections, there will be a quiz that'll test your knowledge on a bunch of sections at once. These quizzes are found in `exercises/quizN.rs`.

Enabling rust-analyzer

Run the command `rustlings lsp` which will generate a `rust-project.json` at the root of the project, this allows [rust-analyzer](#) to parse each exercise.

Continuing On

Once you've completed Rustlings, put your new knowledge to good use! Continue practicing your Rust skills by building your own projects, contributing to Rustlings, or finding other open-source projects to contribute to.

Uninstalling Rustlings

If you want to remove Rustlings from your system, there are two steps. First, you'll need to remove the exercises folder that the install script created for you:

```
rm -rf rustlings # or your custom folder name, if you chose and or renamed it
```

Second, run `cargo uninstall rustlings` to remove the rustlings binary:

```
cargo uninstall rustlings
```

Now you should be done!

Contributing

See [CONTRIBUTING.md](#).

Development-focused discussion about Rustlings happens in the [rustlings stream](#) on the [Rust Project Zulip](#). Feel free to start a new thread there if you have ideas or suggestions!

Contributors ✨

Thanks goes to the wonderful people listed in [AUTHORS.md](#) 🎉

Variables

In Rust, variables are immutable by default. When a variable is immutable, once a value is bound to a name, you can't change that value. You can make them mutable by adding `mut` in front of the variable name.

Further information

- [Variables and Mutability](#).

```
// variables1.rs
// Make me compile!
// Execute `rustlings hint variables1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    x = 5;
    println!("x has the value {}", x);
}
```

```
// variables2.rs
// Execute `rustlings hint variables2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    let x;
    if x == 10 {
        println!("x is ten!");
    } else {
        println!("x is not ten!");
    }
}
```

Functions

Here, you'll learn how to write functions and how the Rust compiler can help you debug errors even in more complex code.

Further information

- [How Functions Work](#)

```
// functions1.rs
// Execute `rustlings hint functions1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    call_me();
}
```

```
// functions2.rs
// Execute `rustlings hint functions2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    call_me(3);
}
```

```
fn call_me(num:) {
    for i in 0..num {
        println!("Ring! Call number {}", i + 1);
    }
}
```

If

if, the most basic (but still surprisingly versatile!) type of control flow, is what you'll learn here.

Further information

- [Control Flow - if expressions](#)

```
// if1.rs
// Execute `rustlings hint if1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
pub fn bigger(a: i32, b: i32) -> i32 {
    // Complete this function to return the bigger number!
    // Do not use:
    // - another function call
    // - additional variables
}
```

```
// Don't mind this for now :)
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ten_is_bigger_than_eight() {
        assert_eq!(10, bigger(10, 8));
    }

    #[test]
    fn fortytwo_is_bigger_than_thirtytwo() {
        assert_eq!(42, bigger(32, 42));
    }
}
```

```
// if2.rs
```

```
// Step 1: Make me compile!
// Step 2: Get the bar_for_fuzz and default_to_baz tests passing!
// Execute `rustlings hint if2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
pub fn foo_if_fizz(fizzish: &str) -> &str {
    if fizzish == "fizz" {
        "foo"
    } else {
        1
    }
}
```

```
// No test changes needed!
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn foo_for_fizz() {
        assert_eq!(foo_if_fizz("fizz"), "foo")
    }

    #[test]
    fn bar_for_fuzz() {
        assert_eq!(foo_if_fizz("fuzz"), "bar")
    }

    #[test]
    fn default_to_baz() {
        assert_eq!(foo_if_fizz("literally anything"), "baz")
    }
}
```


Primitive Types

Rust has a couple of basic types that are directly implemented into the compiler. In this section, we'll go through the most important ones.

Further information

- [Data Types](#)
- [The Slice Type](#)

```
// primitive_types1.rs
// Fill in the rest of the line that has code missing!
// No hints, there's no tricks, just get used to typing these :)

// I AM NOT DONE

fn main() {
    // Booleans (`bool`)

    let is_morning = true;
    if is_morning {
        println!("Good morning!");
    }

    let // Finish the rest of this line like the example! Or make it be false!
    if is_evening {
        println!("Good evening!");
    }
}
```

```
// primitive_types2.rs
// Fill in the rest of the line that has code missing!
// No hints, there's no tricks, just get used to typing these :)

// I AM NOT DONE

fn main() {
    // Characters (`char`)

    // Note the _single_ quotes, these are different from the double quotes
    // you've been seeing around.
    let my_first_initial = 'C';
    if my_first_initial.is_alphabetic() {
        println!("Alphabetical!");
    } else if my_first_initial.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }

    let // Finish this line like the example! What's your favorite character?
    // Try a letter, try a number, try a special character, try a character
    // from a different language than your own, try an emoji!
    if your_character.is_alphabetic() {
        println!("Alphabetical!");
    } else if your_character.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }
}
```

Vectors

Vectors are one of the most-used Rust data structures. In other programming languages, they'd simply be called Arrays, but since Rust operates on a bit of a lower level, an array in Rust is stored on the stack (meaning it can't grow or shrink, and the size needs to be known at compile time), and a Vector is stored in the heap (where these restrictions do not apply).

Vectors are a bit of a later chapter in the book, but we think that they're useful enough to talk about them a bit earlier. We shall be talking about the other useful data structure, hash maps, later.

Further information

- [Storing Lists of Values with Vectors](#)
- [iter_mut](#)
- [map](#)

```
// vecs1.rs
// Your task is to create a `Vec` which holds the exact same elements
// as in the array `a`.
// Make me compile and pass the test!
// Execute `rustlings hint vecs1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn array_and_vec() -> ([i32; 4], Vec<i32>) {
    let a = [10, 20, 30, 40]; // a plain array
    let v = // TODO: declare your vector here with the macro for vectors

    (a, v)
}
```

```
#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn test_array_and_vec_similarity() {
        let (a, v) = array_and_vec();
        assert_eq!(a, v[..]);
    }
}
```

```
// vecs2.rs
// A Vec of even numbers is given. Your task is to complete the loop
// so that each number in the Vec is multiplied by 2.
//
// Make me pass the test!
//
// Execute `rustlings hint vecs2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn vec_loop(mut v: Vec<i32>) -> Vec<i32> {
    for element in v.iter_mut() {
        // TODO: Fill this up so that each element in the Vec `v` is
        // multiplied by 2.
        ???
    }
}
```

```
// At this point, `v` should be equal to [4, 8, 12, 16, 20].
v
```

```
fn vec_map(v: &Vec<i32>) -> Vec<i32> {
    v.iter().map(|element| {
        // TODO: Do the same thing as above - but instead of mutating the
        // Vec, you can just return the new number!
        ???
    }).collect()
}
```


Move Semantics

These exercises are adapted from [pnkfelix's Rust Tutorial](#) -- Thank you Felix!!!

Further information

For this section, the book links are especially important.

- [Ownership](#)
- [Reference and borrowing](#)

```
// move_semantics1.rs
// Execute `rustlings hint move_semantics1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

fn main() {
    let vec0 = Vec::new();

    let vec1 = fill_vec(vec0);

    println!("{}", "vec1", vec1.len(), vec1);

    vec1.push(88);

    println!("{}", "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}

// move_semantics2.rs
// Execute `rustlings hint move_semantics2` or use the `hint` watch subcommand for a hint.

// Expected output:
// vec0 has length 3 content `[22, 44, 66]`
// vec1 has length 4 content `[22, 44, 66, 88]`

// I AM NOT DONE

fn main() {
    let vec0 = Vec::new();

    // Do not move the following line!
    let mut vec1 = fill_vec(vec0);

    // Do not change the following line!
    println!("{}", "vec0", vec0.len(), vec0);

    vec1.push(88);

    println!("{}", "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);
```


Structs

Rust has three struct types: a classic C struct, a tuple struct, and a unit struct.

Further information

- [Structures](#)
- [Method Syntax](#)

```
// structs1.rs
// Address all the TODOs to make the tests pass!
// Execute `rustlings hint structs1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
struct ColorClassicStruct {
    // TODO: Something goes here
}
```

```
struct ColorTupleStruct(/* TODO: Something goes here */);
```

```
#[derive(Debug)]
struct UnitLikeStruct;
```

```
#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn classic_c_structs() {
        // TODO: Instantiate a classic c struct!
        // let green =
```

```
        assert_eq!(green.red, 0);
        assert_eq!(green.green, 255);
        assert_eq!(green.blue, 0);
    }
```

```
    #[test]
    fn tuple_structs() {
        // TODO: Instantiate a tuple struct!
        // let green =
```

```
        assert_eq!(green.0, 0);
        assert_eq!(green.1, 255);
        assert_eq!(green.2, 0);
    }
```

```
    #[test]
    fn unit_structs() {
        // TODO: Instantiate a unit-like struct!
        // let unit_like_struct =
        let message = format!("{:?}s are fun!", unit_like_struct);
```

```
        assert_eq!(message, "UnitLikeStructs are fun!");
    }
```

```
}
```

```
// structs2.rs
// Address all the TODOs to make the tests pass!
// Execute `rustlings hint structs2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
#[derive(Debug)]
struct Order {
    name: String,
    year: u32,
    made_by_phone: bool,
```

```
    made_by_mobile: bool,
    made_by_email: bool,
    item_number: u32,
    count: u32,
}

fn create_order_template() -> Order {
    Order {
        name: String::from("Bob"),
        year: 2019,
        made_by_phone: false,
        made_by_mobile: false,
        made_by_email: true,
        item_number: 123,
        count: 0,
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn your_order() {
        let order_template = create_order_template();
        // TODO: Create your own order using the update syntax and template above!
        // let your_order =
        assert_eq!(your_order.name, "Hacker in Rust");
        assert_eq!(your_order.year, order_template.year);
        assert_eq!(your_order.made_by_phone, order_template.made_by_phone);
        assert_eq!(your_order.made_by_mobile, order_template.made_by_mobile);
        assert_eq!(your_order.made_by_email, order_template.made_by_email);
        assert_eq!(your_order.item_number, order_template.item_number);
        assert_eq!(your_order.count, 1);
    }
}
```

Enums

Rust allows you to define types called "enums" which enumerate possible values. Enums are a feature in many languages, but their capabilities differ in each language. Rust's enums are most similar to algebraic data types in functional languages, such as F#, OCaml, and Haskell. Useful in combination with enums is Rust's "pattern matching" facility, which makes it easy to run different code for different values of an enumeration.

Further information

- [Enums](#)
- [Pattern syntax](#)

```
// enums1.rs
// No hints this time! ;)

// I AM NOT DONE

#[derive(Debug)]
enum Message {
    // TODO: define a few types of messages as used below
}

fn main() {
    println!("{:?}", Message::Quit);
    println!("{:?}", Message::Echo);
    println!("{:?}", Message::Move);
    println!("{:?}", Message::ChangeColor);
}

// enums2.rs
// Execute `rustlings hint enums2` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

#[derive(Debug)]
enum Message {
    // TODO: define the different variants used below
}

impl Message {
    fn call(&self) {
        println!("{:?}", self);
    }
}

fn main() {
    let messages = [
        Message::Move { x: 10, y: 30 },
        Message::Echo(String::from("hello world")),
        Message::ChangeColor(200, 255, 255),
        Message::Quit,
    ];

    for message in &messages {
        message.call();
    }
}
```


Strings

Rust has two string types, a string slice (&str) and an owned string (String). We're not going to dictate when you should use which one, but we'll show you how to identify and create them, as well as use them.

Further information

- [Strings](#)

```
// strings1.rs
// Make me compile without changing the function signature!
// Execute `rustlings hint strings1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    let answer = current_favorite_color();
    println!("My current favorite color is {}", answer);
}
```

```
fn current_favorite_color() -> String {
    "blue"
}
```

```
// strings2.rs
// Make me compile without changing the function signature!
// Execute `rustlings hint strings2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    let word = String::from("green"); // Try not changing this line :)
    if is_a_color_word(word) {
        println!("That is a color word I know!");
    } else {
        println!("That is not a color word I know.");
    }
}
```

```
fn is_a_color_word(attempt: &str) -> bool {
    attempt == "green" || attempt == "blue" || attempt == "red"
}
```

Modules

In this section we'll give you an introduction to Rust's module system.

Further information

- [The Module System](#)

```
// modules1.rs
// Execute `rustlings hint modules1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

mod sausage_factory {
    // Don't let anybody outside of this module see this!
    fn get_secret_recipe() -> String {
        String::from("Ginger")
    }

    fn make_sausage() {
        get_secret_recipe();
        println!("sausage!");
    }
}

fn main() {
    sausage_factory::make_sausage();
}


// modules2.rs
// You can bring module paths into scopes and provide new names for them with the
// 'use' and 'as' keywords. Fix these 'use' statements to make the code compile.
// Execute `rustlings hint modules2` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

mod delicious_snacks {
    // TODO: Fix these use statements
    use self::fruits::PEAR as ???
    use self::veggies::CUCUMBER as ???

    mod fruits {
        pub const PEAR: &'static str = "Pear";
        pub const APPLE: &'static str = "Apple";
    }

    mod veggies {
        pub const CUCUMBER: &'static str = "Cucumber";
        pub const CARROT: &'static str = "Carrot";
    }
}

fn main() {
    println!(
        "favorite snacks: {} and {}",
        delicious_snacks::fruit,
        delicious_snacks::veggie
    );
}
```

Hashmaps

A *hash map* allows you to associate a value with a particular key. You may also know this by the names [unordered map in C++](#), [dictionary in Python](#) or an *associative array* in other languages.

This is the other data structure that we've been talking about before, when talking about Vecs.

Further information

- [Storing Keys with Associated Values in Hash Maps](#)

```
// hashmaps1.rs
// A basket of fruits in the form of a hash map needs to be defined.
// The key represents the name of the fruit and the value represents
// how many of that particular fruit is in the basket. You have to put
// at least three different types of fruits (e.g apple, banana, mango)
// in the basket and the total count of all the fruits should be at
// least five.
//
// Make me compile and pass the tests!
//
// Execute `rustlings hint hashmaps1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE
```

```
use std::collections::HashMap;
```

```
fn fruit_basket() -> HashMap<String, u32> {
    let mut basket = // TODO: declare your hash map here.

    // Two bananas are already given for you :)
    basket.insert(String::from("banana"), 2);

    // TODO: Put more fruits in your basket here.

    basket
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn at_least_three_types_of_fruits() {
        let basket = fruit_basket();
        assert!(basket.len() >= 3);
    }

    #[test]
    fn at_least_five_fruits() {
        let basket = fruit_basket();
        assert!(basket.values().sum::<u32>() >= 5);
    }
}
```

```
// hashmaps2.rs
// We're collecting different fruits to bake a delicious fruit cake.
// For this, we have a basket, which we'll represent in the form of a hash
// map. The key represents the name of each fruit we collect and the value
// represents how many of that particular fruit we have collected.
// Three types of fruits - Apple (4), Mango (2) and Lychee (5) are already
// in the basket hash map.
// You must add fruit to the basket so that there is at least
// one of each kind and more than 11 in total - we have a lot of mouths to feed.
// You are not allowed to insert any more of these fruits!
//
// Make me pass the tests!
//
// Execute `rustlings hint hashmaps2` or use the `hint` watch subcommand for a hint.
```


Options

Type Option represents an optional value: every Option is either Some and contains a value, or None, and does not. Option types are very common in Rust code, as they have a number of uses:

- Initial values
- Return values for functions that are not defined over their entire input range (partial functions)
- Return value for otherwise reporting simple errors, where None is returned on error
- Optional struct fields
- Struct fields that can be loaned or "taken"
- Optional function arguments
- Nullable pointers
- Swapping things out of difficult situations

Further Information

- [Option Enum Format](#)
- [Option Module Documentation](#)
- [Option Enum Documentation](#)
- [if let](#)
- [while let](#)

```
// options1.rs
// Execute `rustlings hint options1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

// This function returns how much icecream there is left in the fridge.
// If it's before 10PM, there's 5 pieces left. At 10PM, someone eats them
// all, so there'll be no more left :(
fn maybe_icecream(time_of_day: u16) -> Option<u16> {
    // We use the 24-hour system here, so 10PM is a value of 22 and 12AM is a value of 0
    // The Option output should gracefully handle cases where time_of_day > 23.
    // TODO: Complete the function body - remember to return an Option!
    ???
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn check_icecream() {
        assert_eq!(maybe_icecream(9), Some(5));
        assert_eq!(maybe_icecream(10), Some(5));
        assert_eq!(maybe_icecream(23), Some(0));
        assert_eq!(maybe_icecream(22), Some(0));
        assert_eq!(maybe_icecream(25), None);
    }

    #[test]
    fn raw_value() {
        // TODO: Fix this test. How do you get at the value contained in the Option?
        let icecreams = maybe_icecream(12);
        assert_eq!(icecreams, 5);
    }
}
```

```
// options2.rs
// Execute `rustlings hint options2` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

#[cfg(test)]
mod tests {
    #[test]
```


Error handling

Most errors aren't serious enough to require the program to stop entirely. Sometimes, when a function fails, it's for a reason that you can easily interpret and respond to. For example, if you try to open a file and that operation fails because the file doesn't exist, you might want to create the file instead of terminating the process.

Further information

- [Error Handling](#)
- [Generics](#)
- [Result](#)
- [Boxing errors](#)

```
// errors1.rs
// This function refuses to generate text to be printed on a nametag if
// you pass it an empty string. It'd be nicer if it explained what the problem
// was, instead of just sometimes returning `None`. Thankfully, Rust has a similar
// construct to `Option` that can be used to express error conditions. Let's use it!
// Execute `rustlings hint errors1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
pub fn generate_nametag_text(name: String) -> Option<String> {
    if name.is_empty() {
        // Empty names aren't allowed.
        None
    } else {
        Some(format!("Hi! My name is {}", name))
    }
}
```

```
#[cfg(test)]
```

```
mod tests {
```

```
    use super::*;
```

```
    #[test]
```

```
    fn generates_nametag_text_for_a_nonempty_name() {
```

```
        assert_eq!(
```

```
            generate_nametag_text("Beyoncé".into()),
```

```
            Ok("Hi! My name is Beyoncé".into())
```

```
        );
```

```
    }
```

```
    #[test]
```

```
    fn explains_why_generating_nametag_text_fails() {
```

```
        assert_eq!(
```

```
            generate_nametag_text("").into()),
```

```
            // Don't change this line
```

```
            Err("`name` was empty; it must be nonempty.".into())
```

```
        );
```

```
    }
```

```
}
```

```
// errors2.rs
```

```
// Say we're writing a game where you can buy items with tokens. All items cost
```

```
// 5 tokens, and whenever you purchase items there is a processing fee of 1
```

```
// token. A player of the game will type in how many items they want to buy,
```

```
// and the `total_cost` function will calculate the total cost of the tokens.
```

```
// Since the player typed in the quantity, though, we get it as a string-- and
```

```
// they might have typed anything, not just numbers!
```

```
// Right now, this function isn't handling the error case at all (and isn't
```

```
// handling the success case properly either). What we want to do is:
```

```
// if we call the `parse` function on a string that is not a number, that
```

```
// function will return a `ParseIntError`, and in that case, we want to
```

```
// immediately return that error from our function and not try to multiply
```

```
// and add.
```


Generics

Generics is the topic of generalizing types and functionalities to broader cases. This is extremely useful for reducing code duplication in many ways, but can call for rather involving syntax. Namely, being generic requires taking great care to specify over which types a generic type is actually considered valid. The simplest and most common use of generics is for type parameters.

Further information

- [Generic Data Types](#)
- [Bounds](#)

```
// This shopping list program isn't compiling!
// Use your knowledge of generics to fix it.

// Execute `rustlings hint generics1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

fn main() {
    let mut shopping_list: Vec<?> = Vec::new();
    shopping_list.push("milk");
}

// This powerful wrapper provides the ability to store a positive integer value.
// Rewrite it using generics so that it supports wrapping ANY type.

// Execute `rustlings hint generics2` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

struct Wrapper {
    value: u32,
}

impl Wrapper {
    pub fn new(value: u32) -> Self {
        Wrapper { value }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn store_u32_in_wrapper() {
        assert_eq!(Wrapper::new(42).value, 42);
    }

    #[test]
    fn store_str_in_wrapper() {
        assert_eq!(Wrapper::new("Foo").value, "Foo");
    }
}
```

Traits

A trait is a collection of methods.

Data types can implement traits. To do so, the methods making up the trait are defined for the data type. For example, the `String` data type implements the `From<str>` trait. This allows a user to write `String::from("hello")`.

In this way, traits are somewhat similar to Java interfaces and C++ abstract classes.

Some additional common Rust traits include:

- `Clone` (the `clone` method)
- `Display` (which allows formatted display via `{}`)
- `Debug` (which allows formatted display via `{:?}`)

Because traits indicate shared behavior between data types, they are useful when writing generics.

Further information

- [Traits](#)

```
// traits1.rs
// Time to implement some traits!
//
// Your task is to implement the trait
// `AppendBar` for the type `String`.
//
// The trait AppendBar has only one function,
// which appends "Bar" to any object
// implementing this trait.
// Execute `rustlings hint traits1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
trait AppendBar {
    fn append_bar(self) -> Self;
}
```

```
impl AppendBar for String {
    // TODO: Implement `AppendBar` for type `String`.
}
```

```
fn main() {
    let s = String::from("Foo");
    let s = s.append_bar();
    println!("s: {}", s);
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn is_foo_bar() {
        assert_eq!(String::from("Foo").append_bar(), String::from("FooBar"));
    }
}
```

```
#[test]
fn is_bar_bar() {
    assert_eq!(
        String::from("").append_bar().append_bar(),
        String::from("BarBar")
    );
}
```

```
// traits2.rs
//
// Your task is to implement the trait
```


Tests

Going out of order from the book to cover tests -- many of the following exercises will ask you to make tests pass!

Further information

- [Writing Tests](#)

```
// tests1.rs
// Tests are important to ensure that your code does what you think it should do.
// Tests can be run on this file with the following command:
// rustlings run tests1
```

```
// This test has a problem with it -- make the test compile! Make the test
// pass! Make the test fail!
// Execute `rustlings hint tests1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
#[cfg(test)]
mod tests {
    #[test]
    fn you_can_assert() {
        assert!();
    }
}
```

```
// tests2.rs
// This test has a problem with it -- make the test compile! Make the test
// pass! Make the test fail!
// Execute `rustlings hint tests2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
#[cfg(test)]
mod tests {
    #[test]
    fn you_can_assert_eq() {
        assert_eq!();
    }
}
```

Lifetimes

Lifetimes tell the compiler how to check whether references live long enough to be valid in any given situation. For example lifetimes say "make sure parameter 'a' lives as long as parameter 'b' so that the return value is valid".

They are only necessary on borrows, i.e. references, since copied parameters or moves are owned in their scope and cannot be referenced outside. Lifetimes mean that calling code of e.g. functions can be checked to make sure their arguments are valid. Lifetimes are restrictive of their callers.

If you'd like to learn more about lifetime annotations, the [lifetimekata](#) project has a similar style of exercises to Rustlings, but is all about learning to write lifetime annotations.

Further information

- [Lifetimes \(in Rust By Example\)](#).
- [Validating References with Lifetimes](#)

```
// lifetimes1.rs
//
// The Rust compiler needs to know how to check whether supplied references are
// valid, so that it can let the programmer know if a reference is at risk
// of going out of scope before it is used. Remember, references are borrows
// and do not own their own data. What if their owner goes out of scope?
//
// Execute `rustlings hint lifetimes1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is '{}'", result);
}
```

```
// lifetimes2.rs
//
// So if the compiler is just validating the references passed
// to the annotated parameters and the return type, what do
// we need to change?
//
// Execute `rustlings hint lifetimes2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
}
```


Iterators

This section will teach you about Iterators.

Further information

- [Iterator](#)
- [Iterator documentation](#)

```
// iterators1.rs
//
// Make me compile by filling in the `???`s
//
// When performing operations on elements within a collection, iterators are essential.
// This module helps you get familiar with the structure of using an iterator and
// how to go through elements within an iterable collection.
//
// Execute `rustlings hint iterators1` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

fn main() {
    let my_fav_fruits = vec!["banana", "custard apple", "avocado", "peach", "raspberry"];

    let mut my_iterable_fav_fruits = ???;    // TODO: Step 1

    assert_eq!(my_iterable_fav_fruits.next(), Some(&"banana"));
    assert_eq!(my_iterable_fav_fruits.next(), ???);    // TODO: Step 2
    assert_eq!(my_iterable_fav_fruits.next(), Some(&"avocado"));
    assert_eq!(my_iterable_fav_fruits.next(), ???);    // TODO: Step 3
    assert_eq!(my_iterable_fav_fruits.next(), Some(&"raspberry"));
    assert_eq!(my_iterable_fav_fruits.next(), ???);    // TODO: Step 4
}

// iterators2.rs
// In this exercise, you'll learn some of the unique advantages that iterators
// can offer. Follow the steps to complete the exercise.
// Execute `rustlings hint iterators2` or use the `hint` watch subcommand for a hint.

// I AM NOT DONE

// Step 1.
// Complete the `capitalize_first` function.
// "hello" -> "Hello"
pub fn capitalize_first(input: &str) -> String {
    let mut c = input.chars();
    match c.next() {
        None => String::new(),
        Some(first) => ???,
    }
}

// Step 2.
// Apply the `capitalize_first` function to a slice of string slices.
// Return a vector of strings.
// ["hello", "world"] -> ["Hello", "World"]
pub fn capitalize_words_vector(words: &[&str]) -> Vec<String> {
    vec![]
}

// Step 3.
// Apply the `capitalize_first` function again to a slice of string slices.
// Return a single string.
// ["hello", " ", "world"] -> "Hello World"
pub fn capitalize_words_string(words: &[&str]) -> String {
    String::new()
}

#[cfg(test)]
```


Threads

In most current operating systems, an executed program's code is run in a process, and the operating system manages multiple processes at once. Within your program, you can also have independent parts that run simultaneously. The features that run these independent parts are called threads.

Further information

- [Dining Philosophers example](#)
- [Using Threads to Run Code Simultaneously](#)

```
// threads1.rs
// Execute `rustlings hint threads1` or use the `hint` watch subcommand for a hint.

// This program spawns multiple threads that each run for at least 250ms,
// and each thread returns how much time they took to complete.
// The program should wait until all the spawned threads have finished and
// should collect their return values into a vector.

// I AM NOT DONE

use std::thread;
use std::time::{Duration, Instant};

fn main() {
    let mut handles = vec![];
    for i in 0..10 {
        handles.push(thread::spawn(move || {
            let start = Instant::now();
            thread::sleep(Duration::from_millis(250));
            println!("thread {} is complete", i);
            start.elapsed().as_millis()
        })));
    }

    let mut results: Vec<u128> = vec![];
    for handle in handles {
        // TODO: a struct is returned from thread::spawn, can you use it?
    }

    if results.len() != 10 {
        panic!("Oh no! All the spawned threads did not finish!");
    }

    println!();
    for (i, result) in results.into_iter().enumerate() {
        println!("thread {} took {}ms", i, result);
    }
}

// threads2.rs
// Execute `rustlings hint threads2` or use the `hint` watch subcommand for a hint.
// Building on the last exercise, we want all of the threads to complete their work but this time
// the spawned threads need to be in charge of updating a shared value: JobStatus.jobs_completed

// I AM NOT DONE

use std::sync::Arc;
use std::thread;
use std::time::Duration;

struct JobStatus {
    jobs_completed: u32,
}

fn main() {
    let status = Arc::new(JobStatus { jobs_completed: 0 });
    let mut handles = vec![];
    for _ in 0..10 {
```


Smart Pointers

In Rust, smart pointers are variables that contain an address in memory and reference some other data, but they also have additional metadata and capabilities. Smart pointers in Rust often own the data they point to, while references only borrow data.

Further Information

- [Smart Pointers](#)
- [Using Box to Point to Data on the Heap](#)
- [Rc<T>, the Reference Counted Smart Pointer](#)
- [Shared-State Concurrency](#)
- [Cow Documentation](#)

```
// arc1.rs
// In this exercise, we are given a Vec of u32 called "numbers" with values ranging
// from 0 to 99 -- [ 0, 1, 2, ..., 98, 99 ]
// We would like to use this set of numbers within 8 different threads simultaneously.
// Each thread is going to get the sum of every eighth value, with an offset.
// The first thread (offset 0), will sum 0, 8, 16, ...
// The second thread (offset 1), will sum 1, 9, 17, ...
// The third thread (offset 2), will sum 2, 10, 18, ...
// ...
// The eighth thread (offset 7), will sum 7, 15, 23, ...
```

```
// Because we are using threads, our values need to be thread-safe. Therefore,
// we are using Arc. We need to make a change in each of the two TODOs.
```

```
// Make this code compile by filling in a value for `shared_numbers` where the
// first TODO comment is, and create an initial binding for `child_numbers`
// where the second TODO comment is. Try not to create any copies of the `numbers` Vec!
// Execute `rustlings hint arc1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
#![forbid(unused_imports)] // Do not change this, (or the next) line.
```

```
use std::sync::Arc;
```

```
use std::thread;
```

```
fn main() {
    let numbers: Vec<_> = (0..100u32).collect();
    let shared_numbers = // TODO
    let mut joinhandles = Vec::new();

    for offset in 0..8 {
        let child_numbers = // TODO
        joinhandles.push(thread::spawn(move || {
            let sum: u32 = child_numbers.iter().filter(|&&n| n % 8 == offset).sum();
            println!("Sum of offset {} is {}", offset, sum);
        }));
    }
    for handle in joinhandles.into_iter() {
        handle.join().unwrap();
    }
}
```

```
// box1.rs
```

```
//
```

```
// At compile time, Rust needs to know how much space a type takes up. This becomes problematic
// for recursive types, where a value can have as part of itself another value of the same type.
// To get around the issue, we can use a `Box` - a smart pointer used to store data on the heap,
// which also allows us to wrap a recursive type.
```

```
//
```

```
// The recursive type we're implementing in this exercise is the `cons list` - a data structure
// frequently found in functional programming languages. Each item in a cons list contains two
// elements: the value of the current item and the next item. The last item is a value called `Nil`.
//
```

```
// Step 1: use a `Box` in the enum definition to make the code compile
```


Macros

Rust's macro system is very powerful, but also kind of difficult to wrap your head around. We're not going to teach you how to write your own fully-featured macros. Instead, we'll show you how to use and create them.

If you'd like to learn more about writing your own macros, the [macrokata](#) project has a similar style of exercises to Rustlings, but is all about learning to write Macros.

Further information

- [Macros](#)
- [The Little Book of Rust Macros](#)

```
// macros1.rs
// Execute `rustlings hint macros1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
macro_rules! my_macro {
    () => {
        println!("Check out my macro!");
    };
}
```

```
fn main() {
    my_macro();
}
```

```
// macros2.rs
// Execute `rustlings hint macros2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    my_macro!();
}
```

```
macro_rules! my_macro {
    () => {
        println!("Check out my macro!");
    };
}
```

Clippy

The Clippy tool is a collection of lints to analyze your code so you can catch common mistakes and improve your Rust code.

If you used the installation script for Rustlings, Clippy should be already installed. If not you can install it manually via `rustup component add clippy`.

Further information

- [GitHub Repository](#).

```
// clippy1.rs
// The Clippy tool is a collection of lints to analyze your code
// so you can catch common mistakes and improve your Rust code.
//
// For these exercises the code will fail to compile when there are clippy warnings
// check clippy's suggestions from the output to solve the exercise.
// Execute `rustlings hint clippy1` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
use std::f32;
```

```
fn main() {
    let pi = 3.14f32;
    let radius = 5.00f32;

    let area = pi * f32::powi(radius, 2);

    println!(
        "The area of a circle with radius {:.2} is {:.5}!",
        radius, area
    )
}
```

```
// clippy2.rs
// Execute `rustlings hint clippy2` or use the `hint` watch subcommand for a hint.
```

```
// I AM NOT DONE
```

```
fn main() {
    let mut res = 42;
    let option = Some(12);
    for x in option {
        res += x;
    }
    println!("{}", res);
}
```


Type conversions

Rust offers a multitude of ways to convert a value of a given type into another type.

The simplest form of type conversion is a type cast expression. It is denoted with the binary operator `as`. For instance, `println!("{}", 1 + 1.0);` would not compile, since `1` is an integer while `1.0` is a float. However, `println!("{}", 1 as f32 + 1.0)` should compile. The exercise [using_as](#) tries to cover this.

Rust also offers traits that facilitate type conversions upon implementation. These traits can be found under the [convert](#) module. The traits are the following:

- `From` and `Into` covered in [from_into](#)
- `TryFrom` and `TryInto` covered in [try_from_into](#)
- `AsRef` and `AsMut` covered in [as_ref_mut](#)

Furthermore, the `std::str` module offers a trait called [FromStr](#) which helps with converting strings into target types via the `parse` method on strings. If properly implemented for a given type `Person`, then `let p: Person = "Mark,20".parse().unwrap();` should both compile and run without panicking.

These should be the main ways ***within the standard library*** to convert data into your desired types.

Further information

These are not directly covered in the book, but the standard library has a great documentation for it.

- [conversions](#)
- [FromStr trait](#)

```
// AsRef and AsMut allow for cheap reference-to-reference conversions.
// Read more about them at https://doc.rust-lang.org/std/convert/trait.AsRef.html
// and https://doc.rust-lang.org/std/convert/trait.AsMut.html, respectively.
// Execute `rustlings hint as_ref_mut` or use the `hint` watch subcommand for a hint.


// I AM NOT DONE


// Obtain the number of bytes (not characters) in the given argument.
// TODO: Add the AsRef trait appropriately as a trait bound.
fn byte_counter<T>(arg: T) -> usize {
    arg.as_ref().as_bytes().len()
}


// Obtain the number of characters (not bytes) in the given argument.
// TODO: Add the AsRef trait appropriately as a trait bound.
fn char_counter<T>(arg: T) -> usize {
    arg.as_ref().chars().count()
}


// Squares a number using as_mut().
// TODO: Add the appropriate trait bound.
fn num_sq<T>(arg: &mut T) {
    // TODO: Implement the function body.
    ???
}


#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn different_counts() {
        let s = "Café au lait";
        assert_ne!(char_counter(s), byte_counter(s));
    }

    #[test]
    fn same_counts() {
        let s = "Cafe au lait";
        assert_eq!(char_counter(s), byte_counter(s));
    }
}
```


Thanks to all contributors of [rustlings](#)!

- The repository of this example is [here](#)
- See other CodeSlide CLI examples [here](#)
- See the installation guide of CodeSlide CLI [here](#)