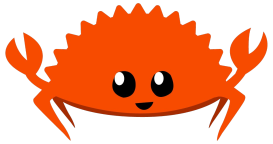
Rust Bootcamp 2021

Day 3



Day 3 Activities

- ✓ More Collections
- ✓ Error Handling
- ✓ Automated Testing
- ✓ Documentation

Collections

The Rust standard library has a few useful collections:

Sequences: Vec, VecDeque, LinkedList

Maps: HashMap, BTreeMap

Sets: HashSet, BTreeSet

BinaryHeap

https://doc.rust-lang.org/std/collections/index.html

Collections - Vector

The vector (Vec) is the most popular collection, and should be the first choice.

```
let v: Vec<i32> = Vec::new();

let v = vec![1, 2, 3];
```

Vectors can be created in two ways.

Note the second version has actual values, so the compiler can infer the type. Further examples will remove the type hint needed on the first line as well.

https://doc.rust-lang.org/book/ch08-01-vectors.html

Collections - Vector

Adding data to a vector is simple:

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

https://doc.rust-lang.org/book/ch08-01-vectors.html

Collections - Vector

Accessing a vector is also easy

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];  // this can panic
println!("The third element is {}", third);

match v.get(2) {  // this is always safe
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

https://doc.rust-lang.org/book/ch08-01-vectors.html

Collections - Vector

But modification after grabbing a reference is an error.

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];

v.push(6);
println!("The first element is: {}", first);
```

The issue is because we use the reference after the modification. Adding an element at the end might reallocate the whole vector, so there is no guarantee the memory referred to still exists.

https://doc.rust-lang.org/book/ch08-01-vectors.html

Collections - Vector

Iterating over a vector is straightforward.

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}

for i in &mut v {
    *i += 50;
}
```

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - Vector

Storing multiple types is possible, but they must be explicitly listed.

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}
let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

https://doc.rust-lang.org/book/ch08-01-vectors.html#using-an-enum-to-store-multiple-types

Collections - VecDeque

A VecDeque is a vector with efficient insertion/deletion on both ends, by storing data in a ring buffer. This makes it ideally suited for queues, especially deques.

It is less efficient than the regular vector for iteration or index operations.

https://doc.rust-lang.org/std/collections/struct.VecDeque.html

Collections - LinkedList

A LinkedList is optimal for splitting and appending lists arbitrarily.

It is much less efficient than the regular vector for everything else, and is **not recommended** for general use. As the docs say:

> "You are *absolutely* certain you *really, truly*, want a doubly linked list."

https://doc.rust-lang.org/std/collections/struct.LinkedList.html

Collections - HashMap

The hashmap is another popular collection, and is designed for key - value storage. It takes two types: HashMap<K, V>

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

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Notice that we need to import the HashMap, as it's not automatically loaded like String or Vec.

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - HashMap

HashMaps own all keys and values, so inserting a value transfers ownership to the map.

```
let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try
// using them and see what compiler error you get!
```

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - HashMap

Access via key is fairly straightforward returning an Option that will be None if the key is not present..

```
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - HashMap

Access via iteration is even easier.

```
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - HashMap

Each key in a HashMap can have only one value. When a value already exists, you can overwrite, ignore the new value, or do a dynamic update.

```
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);
```

Here we overwrite with *insert*, and ignore the new value with *entry* and *or_insert*.

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - HashMap

In this case, we dynamically update an entry based on the previous value, in a simple counter.

```
let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}
```

https://doc.rust-lang.org/book/ch08-03-hash-maps.html

Collections - BTreeMap

A BTreeMap uses a binary tree as the backing data structure, and is a decent choice for sorted maps. This is similar to the regular map in C++.

It is a poor choice for arbitrary lookups, as the HashMap will perform better: O(1) vs O(log n).

https://doc.rust-lang.org/std/collections/struct.BTreeMap.html

Collections - HashSet and BTreeSet

Sets are key-only variants of the maps, with similar properties and usage.

https://doc.rust-lang.org/std/collections/hash_set/struct.HashSet.html

https://doc.rust-lang.org/std/collections/struct.BTreeSet.html

Collections - BinaryHeap

A BinaryHeap is used primarily for a priority queue, where the “largest” elements are always sorted to the top.

Elements already on the heap cannot be modified to change their ordering in-place; they must be removed and added again.

https://doc.rust-lang.org/std/collections/struct.BinaryHeap.html

Error Handling - Panic

The most basic way to handle an error is to exit immediately. This is called a panic in Rust.

```
fn main() {
    panic!("crash and burn");
}
```

The “crash and burn” message will be printed on the console before the program exits. A backtrace is normally not included, but the line the panic occurs on is noted in debug builds.

https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html

Error Handling - Result

Proper error handling can be done with the *Result* type.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

A function that returns a *Result* will either have a proper value, or an error.

Many std library functions return a *Result* because they can fail.

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

Error Handling - Single Error

Here is an example of handling a file open error:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("File error: {:?}", error),
    };
}
```

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

Error Handling - Multiple Errors

```
use std::io::ErrorKind;
let f = match File::open("hello.txt") {
    Ok(file) => file,
    Err(error) => match error.kind() {
        ErrorKind::NotFound => match File::create("hello.txt") {
            Ok(fc) => fc,
            Err(e) => panic!("Problem creating file: {:?}", e),
        },
        other_error => {
            panic!("Problem opening the file: {:?}", other_error)
        }
    },
};
```

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

Error Handling - Multiple Errors

Another style of writing this, using closures to be more concise:

```
let f = File::open("hello.txt").unwrap_or_else(|error| {
    if error.kind() == ErrorKind::NotFound {
        File::create("hello.txt").unwrap_or_else(|error| {
            panic!("Problem creating the file: {:?}", error);
        })
    } else {
        panic!("Problem opening the file: {:?}", error);
    }
});
```

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

Error Handling - Shortcuts

There are some shortcuts for error handling.

```
let f1 = File::open("hello.txt").unwrap();

let f2 = File::open("hello.txt").expect("Failed to open file");
```

The first line uses *unwrap()* to always get the successful result, and panics on failure with the default error message.

The second line uses *expect()* to do the same and also panics, but lets you specify the error message.

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html

Error Handling - Propagating Errors

Instead of *panic*-ing, we can also propagate the error up to the caller.

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = match File::open("hello.txt") {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

h

gating errors

Error Handling - Propagating Errors Shortcut

There’s also a shortcut for propagating errors: the ? operator

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

This does exactly what the match expressions were used for, and returns the error if it exists.

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#a-shortcut-for-propagating-errors-the--operator

Error Handling - Propagating Errors Shortcut

You can also chain together ? statements.

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#a-shortcut-for-propagating-errors-the--operator

Error Handling - Returning errors in main()

In order to use ? inside *main()*, the return type must be changed to a Result

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;

    Ok(())
}
```

https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#the--operator-can-be-used-in-functions-that-return-result

Testing in Rust

A basic unit test looks like this, and goes in the same src file as the code.

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

As a reminder, when you run *cargo test* any tests defined in a project will automatically be run.

https://doc.rust-lang.org/book/ch11-01-writing-tests.html

```
$ cargo test
   Compiling tester v0.1.0 (file:///projects/tester)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
    Running target/debug/deps/tester-92948b65e88960b4

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests tester

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

```
$ cargo test
   Compiling tester v0.1.0 (file:///projects/tester)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
    Running target/debug/deps/tester-92948b65e88960b4

running 2 tests
test tests::another ... FAILED
test tests::it_works ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Testing in Rust - Assert

There are several assert macros to check results:

```
assert!(expr);

assert_eq!(val1, val2);

assert_ne!(val1, val2);
```

https://doc.rust-lang.org/book/ch11-01-writing-tests.html#checking-results-with-the-assert-macro

Testing in Rust - Getting better error messages

The Assert macros take additional arguments to print out in the error case.

```
#[test]
fn greeting_contains_name() {
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{}`",
        result
    );
}
```

https://doc.rust-lang.org/book/ch11-01-writing-tests.html#adding-custom-failure-messages

Testing in Rust - Expecting panics

If you want to test that a code will panic on certain input, that can also be done. Use the special attribute *should_panic*.

```
#[test]
#[should_panic]
fn greater_than_100() {
    Guess::new(200);
}
```

https://doc.rust-lang.org/book/ch11-01-writing-tests.html#checking-for-panics-with-should_panic

Testing in Rust - Expecting panics

You can also test for the expected message of the panic.

```
#[test]
#[should_panic(expected = "Guess value must be <= 100")]
fn greater_than_100() {
    Guess::new(200);
}
```

If the message isn't equal, the test will fail.

https://doc.rust-lang.org/book/ch11-01-writing-tests.html#checking-for-panics-with-should_panic

Testing in Rust - Returning an error

Tests can also return Result<T, E> instead of panic-ing on failure.

```rust
#[test]
fn it_works() -> Result<(), String> {
    if 2 + 2 == 4 {
        Ok(())
    } else {
        Err(String::from("two plus two does not equal four"))
    }
}
```

https://doc.rust-lang.org/book/ch11-01-writing-tests.html#using-resultt-e-in-tests

Testing in Rust - Integration tests

Integration tests live in a special *tests* directory at the top level, next to *src*.

The `#[cfg(test)]` annotation is not needed, as all files and modules in the *tests* directory are considered tests.

Note that for binary crates with *src/main.rs*, you cannot call anything inside there from an integration test, as you cannot import it. You must create a library, with an entrypoint at *src/lib.rs*. A common pattern is to make *src/main.rs* a stub that calls things in *src/lib.rs*.

https://doc.rust-lang.org/book/ch11-03-test-organization.html#integration-tests-for-binary-crates

Documentation

Documentation is built into the src code of Rust, with the triple slash.

```rust
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let answer = my_crate::add_one(5);
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#making-useful-documentation-comments

Documentation

Documentation is built with the rustdoc tool distributed with Rust.

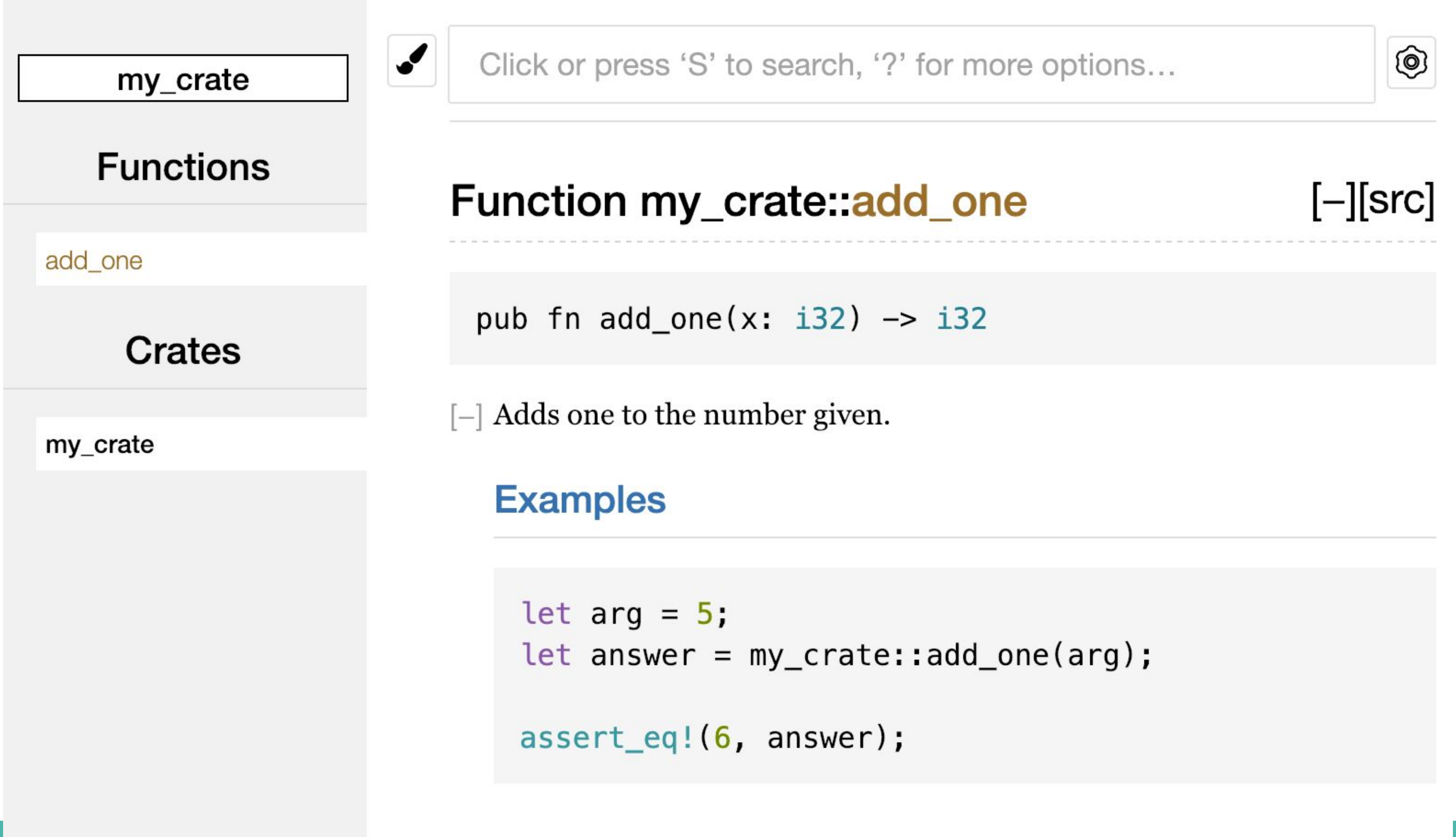
Typically, docs are built with

```
cargo doc
```

For convenience, this will also open a browser window

```
cargo doc --open
```

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#making-useful-documentation-comments



Function my_crate::add_one

```
pub fn add_one(x: i32) -> i32
```

[−] Adds one to the number given.

Examples

```
let arg = 5;
let answer = my_crate::add_one(arg);

assert_eq!(6, answer);
```

Documentation

Common sections in documentation include:

- Examples - example code.
- Panics - reasons why the function might *panic*.
- Errors - if the function returns a *Result*, describing the errors that might occur and what might cause them.
- Safety - if the function is *unsafe* (which it should never be 😎).

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#commonly-used-sections

Documentation as tests

The examples in documentation are automatically run as tests. Recall that we wrote one example, so when we run *cargo test* we also get

```
   Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out
```

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#documentation-comments-as-tests

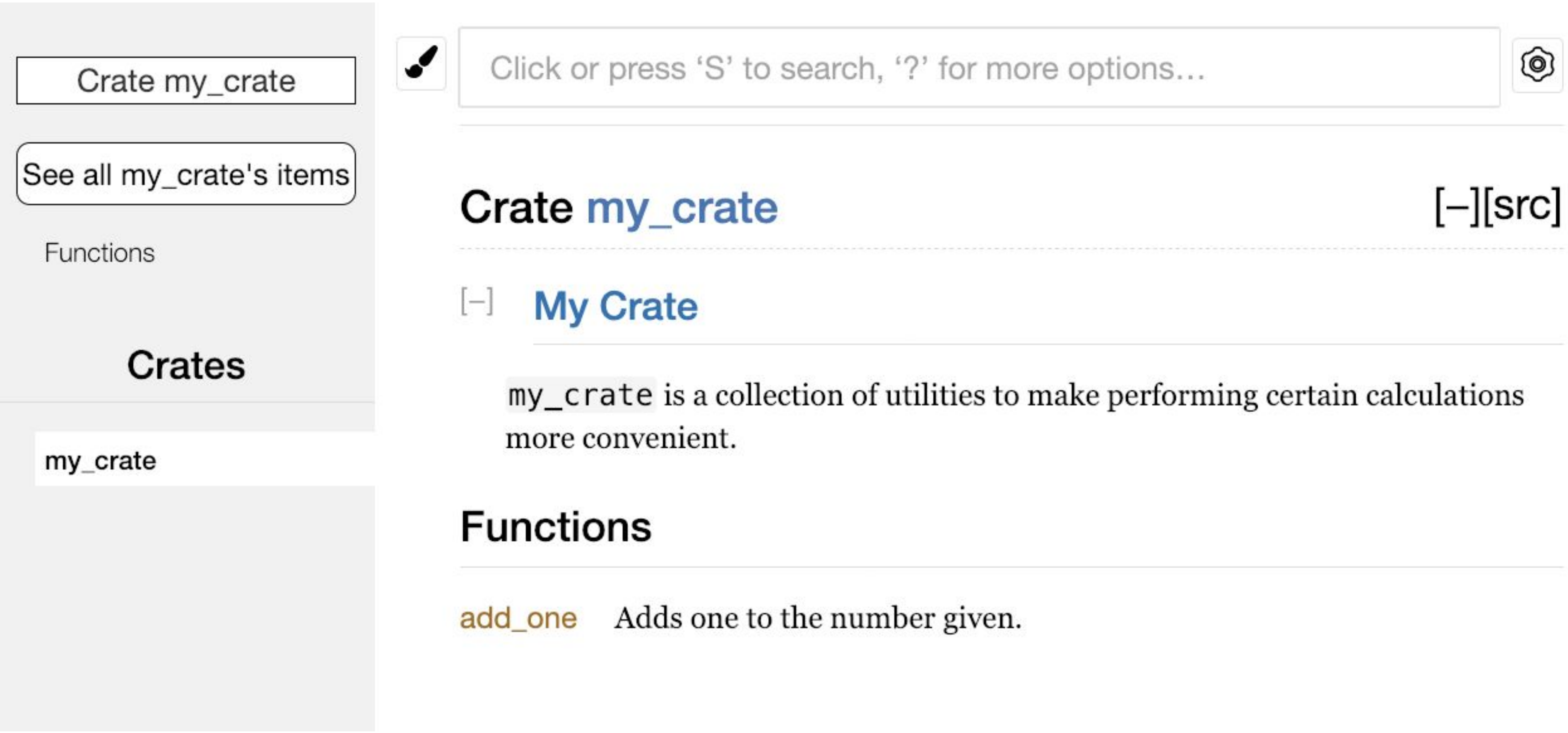
Documentation for the whole library

There is a special comment type for crate-level docs, to be added to *src/lib.rs*

```
//! # My Crate
//!
//! `my_crate` is a collection of utilities to make performing
//! certain calculations more convenient.

// actual code for lib.rs here
```

https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html#commenting-contained-items


Crate my_crate

See all my_crate's items

Functions

Crates

my_crate

Click or press ‘S’ to search, ‘?’ for more options…

Crate my_crate [−][src]

[−] My Crate

`my_crate` is a collection of utilities to make performing certain calculations more convenient.

Functions

add_one Adds one to the number given.

End of Lecture

Exercises

Find and fix the errors in exercises/day3/check_errors

Make tests and documentation for check_errors