Rust Cheatsheet

Abyan Majid, 2023

Cargo

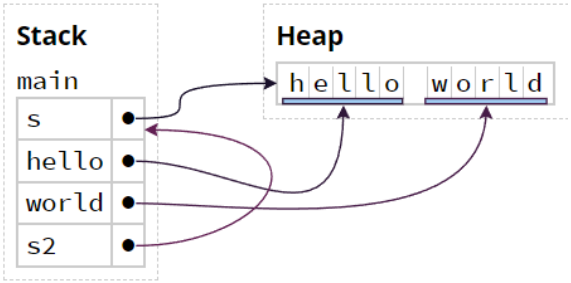
Concept	Snippets/Command/Explanation/Examples
Create a new project	`// binary package` `$ cargo new <project_name>` `// library package` `$ cargo new restaurant --lib`
Compile into an executable	`$ cargo build`
Compile into an executable and then run	`$ cargo run` `// if using cargo workspaces and have multiple binary crates` `$ cargo run -p <binaryCrateName>`
Check if code will compile, without building an executable	`$ cargo check`
Build for release	`$ cargo build --release`
Update dependencies	`$ cargo update`
Run tests	`$ cargo test` `// show program outputs in test run` `$ cargo test -- --show-output` `// evaluate one test function only` `$ cargo test <test_function_name>` `// evaluate all test functions whose names start with a pattern`

	`$ cargo test <pattern>` `// evaluate only one test module` `$ cargo test <test_module_name>`
Build the documentation for your dependencies	`$ cargo doc --open`

Variables	
Concept	**Snippets/Command/Explanation/Examples**
Declare immutable variable	`// can only be declared in functions` `let x = 5;`
Declare mutable variable	`// can only be declared in functions` `let mut x = 5;`
Constants	`// can be in functions or global scope` `const x: u32 = 5;`
Shadowing	`let x = 5;` `let x = "hello";`
Statements vs. Expressions	`// statements end with a semicolon, expressions do not.` `// besides a syntactic scope, {/* */} also denotes an expression.` `let y = {` `  let x = 3; // statement` `  x + 1 // expression` `}; // statement`

Data Types, Collections	
Concept	**Snippets/Command/Explanation/Examples**
(SCALAR) Integer	**Options:** | Length	Signed	Unsigned
---	---	---
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

	Default: i32
(SCALAR) Float	**Options:** f32 (single precision), f64 (double precision) **Default:** f64
(SCALAR) Boolean	**Options:** true, false `let x = true;` `let y: bool = false; // with explicit type annotation`
(SCALAR) Char	`// use single quotes` `let c = 'z';` `let z: char = 'ℤ'; // with explicit type annotation`
(COMPOUND) Tuple	`// fixed size; cannot grow or shrink, the elements may have different types` `let tup: (i32, f64, u8) = (500, 6.4, 1);` `// destructure a tuple` `let (x, y, z) = tup;` `// indexing tuples` `let a = tup.0; // a = 500` `// empty tuple “()” is called a unit`
(COMPOUND) Array	`// fixed size; cannot grow or shrink, the elements must have the same type` `let arr = [1, 2, 3, 4, 5];` `// with type and length annotation` `let arr: [i32; 5] = [1, 2, 3, 4, 5]; // type: i32, length: 5` `// initialise an array of the same values` `let arr = [3; 5]; // the same as arr = [3, 3, 3, 3, 3]` `// indexing arrays` `let first = arr[0]'`
(COMPOUND) String	`// create empty `String`` `let mut a = String::new();`  `// create `String` with initial value` `let b = String::from("hello, world!");`  `// initialise a string slice `str`` `let c: &str = "hello, world!";` `// convert a string slice `str` to `String` type` `let d: &str = "hello, world!".to_string();` `// appending a string slice to `String`` `let mut s = String::from("foo");` `s.push_str("bar");` `// concatenation with + (calls the standard add method)` `// fn add(self, s: &str) -> String` `// hence why only s2 is referenced`

	``rust let s1 = String::from("Hello, "); let s2 = String::from("world!"); let s3 = s1 + &s2; // !format macro let s1 = String::from("tic"); let s2 = String::from("tac"); let s3 = String::from("toe"); let s = format!("{s1}-{s2}-{s3}"); // iterating over strings by characters for c in "3∂".chars() { println!("{c}"); } // iterating over strings by bytes for b in "3∂".bytes() { println!("{b}"); } ```
Slicing strings	```rust // rust doesn't support indexing, slice instead. let s = String::from("hello world"); let hello: &str = &s[0..5]; let world: &str = &s[6..11]; let s2: &String = &s; ``` 
Struct	```rust // defining a struct struct User { active: bool, username: String, email: String, sign_in_count: u64, } // creating a User instance fn main() { let user1 = User { email: String::from("someone@example.com"), username: String::from("someusername123"), active: true, sign_in_count: 1, }; } // struct update syntax let user2 = User { email: String::from("another@example.com"), ```

	`` ..user1 // inherit every other field from instance `user1` }; // tuple struct struct RGB(i32, i32, i32); struct Point(i32, i32, i32); fn main() { let black = RGB(0, 0, 0); let origin = Point(0, 0, 0); } // unit-like struct (no fields) struct UnitLikeStruct; fn main() { let subject = UnitLikeStruct; } ```
Enum	``` // enum without associated data enum IpAddrKind { V4, V6, } struct IpAddr { kind: IpAddrKind, address: String, } let home = IpAddr { kind: IpAddrKind::V4, address: String::from("127.0.0.1"), }; // enum with associated data enum IpAddr { V4(u8, u8, u8, u8), V6(String), } let home = IpAddr::V4(127, 0, 0, 1); let loopback = IpAddr::V6(String::from("::1")); // special `Option<T>` enum to denote presence or absence of value enum Option<T> { None, Some(T), } // `T` is type let some_number = Some(5); let some_char = Some('e'); let absent_number: Option<i32> = None; ```
Vector	``` // get returns Option<T>, so it doesn't panic and you should handle None. let v = vec![1, 2, 3, 4, 5]; ```

	``rust let third: &i32 = &v[2]; // via indexing let third: Option<&i32> = v.get(2); // via `get`, returns Option<T> match third { Some(third) => println!("The third element is {third}"), None => println!("There is no third element.") } // iterating over immutable vector let v = vec![100, 32, 57]; for n_ref in &v { // n_ref has type &i32 let n_plus_one: i32 = *n_ref + 1; println!("{n_plus_one}"); } // iterating over mutable vector and changing the elements let mut v = vec![100, 32, 57]; for n_ref in &mut v { // n_ref has type &mut i32 *n_ref += 50; } // using enum to store multiple types enum SpreadsheetCell { Int(i32), Float(f64), Text(String), } let row = vec![SpreadsheetCell::Int(3), SpreadsheetCell::Text(String::from("blue")), SpreadsheetCell::Float(10.12),]; ```
Hash map	```rust // bring into scope from the std module use std::collections::HashMap; // create an empty hashmap let mut scores = HashMap::new(); // add key-value pairs with `insert` // all keys must have the same type, and all values must have the same type // NOTE: insert will overwrite a key-value pair if the key already exists scores.insert(String::from("Blue"), 10); scores.insert(String::from("Yellow"), 50); // adding a key-value pair only if key doesn’t exist scores.entry(String::from("Yellow")).or_insert(50); // retrieve value from hash map // get value by key "Blue", if None, assign 0 by default. // copied() returns an Option<T> instead of Option<&T> let score = scores.get("Blue").copied().unwrap_or(0) // iterate over a hashmap ```

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

Numeric Operations	
Concept	**Snippets/Command/Explanation/Examples**
Addition	`let sum = 5 + 10;`
Subtraction	`let difference = 95.5 - 4.3;`
Multiplication	`let product = 4 * 30;`
Division	`// division truncates towards zero` `let quotient = 56.7 / 32.2;` `let truncated = -5 / 3; // Results in -1`
Remainder	`let remainder = 43 % 5;`

Function & Method Syntaxes	
Concept	**Snippets/Command/Explanation/Examples**
Function	```rust // if not a return type is not specified, all functions default to returning a unit/empty tuple ie. () fn plus_one(arg:i32) -> i32 { return arg + 1; } fn main() { let x = plus_one(5); println!("The value of x is {x}."); // The value of x is 6. } ```
Method Syntax	```rust struct Rectangle { width: u32, height: u32, } // `impl` is a keyword to denote methods defined on the Rectangle struct impl Rectangle { fn area(&self) -> u32 { self.width * self.height } // method that accepts another Rectangle instance fn can_hold(&self, other: &Rectangle) -> bool { self.width > other.width && self.height > other.height } } ```

	`` fn main() { let rect1 = Rectangle { width: 30, height: 50, }; println!("The area of the rectangle is {} square pixels.", rect1.area() // call method `area`); } // IMPL CAN ALSO BE DONE ON AN ENUM ```

Control Flow	
Concept	**Snippets/Command/Explanation/Examples**
if-else if-else	``` fn main() { let number = 6; if number % 4 == 0 { println!("number is divisible by 4"); } else if number % 3 == 0 { println!("number is divisible by 3"); } else if number % 2 == 0 { println!("number is divisible by 2"); } else { println!("number is not divisible by 4, 3, or 2"); } } ```
Inline if-else	``` fn main() { let condition = true; let number = if condition { 5 } else { 6 }; // types must be the same } ```
if-let	``` // USEFUL FOR ENUMS // with Option<T> let config_max = Some(3u8); if let Some(max) = config_max { println!("The maximum is configured to be {}", max); } // with other enums let mut count = 0; if let Coin::Quarter(state) = coin { println!("State quarter from {:?}!", state); } else { count += 1; } ```

loop (Infinite)

```rust
// loops indefinitely - the only way to exit is through a break statement
(you can also write a value to return beside the break statement).
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };
    println!("The result is {result}");
}

// break from a parent loop through labels
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

while loop

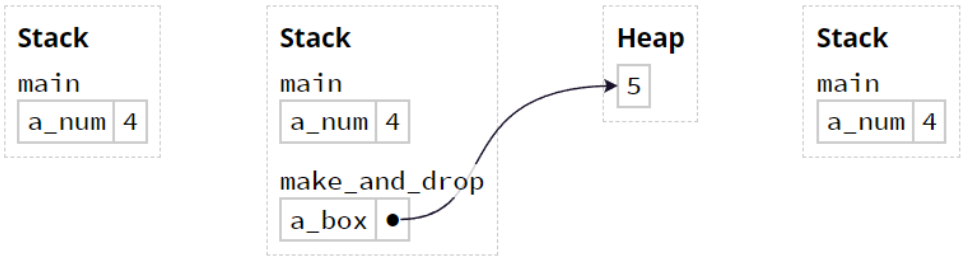
```rust
// Loop for as long as a condition is true
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

for loop

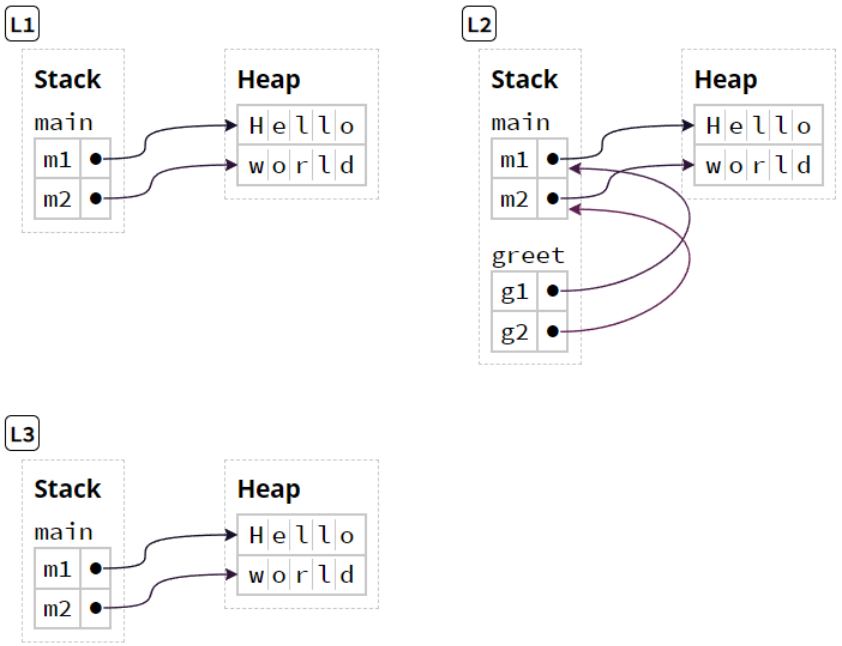
```rust
// iterate over a collection
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
```

	``rust println!("the value is: {element}"); } } // loop for a specified number of times fn main() { let a = [10, 20, 30, 40, 50]; for element in a { println!("the value is: {element}"); } } fn main() { // 1..4 is exclusive of the upper bound, so it counts from 1 to 3. // ascending count for number in 1..4 { println!("{number}!"); } // descending count for number in (1..4).rev() { println!("{number}") } } ```
match	```rust enum Coin { Penny, Nickel, Dime, Quarter, } fn value_in_cents(coin: Coin) -> u8 { match coin { Coin::Penny => { println!("Lucky penny!"); 1 } Coin::Nickel => 5, Coin::Dime => 10, Coin::Quarter => 25, _ => { // `_` catches every other pattern (default state) println!("Default: No coin!") } } } OTHER PATTERNS: // `|` matches multiple values // `1..=5` matches an inclusive range from 1 to 5 ```

Ownership, References, Borrowing	
Concept	**Snippets/Command/Explanation/Examples**

Box deallocation principle	If a variable owns a box, when Rust deallocates the variable's frame, then Rust deallocates the box's heap memory. (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Moved heap data principle	If a variable x moves ownership of heap data to another variable y, then x cannot be used after the move. (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Pointer safety principle	Data should never be aliased and mutated at the same time. (https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html)
Put data on the heap with Box + basic ownership	`fn main() {` `  let a = Box::new([0; 1_000_000]); // a owns the array` `  let b = a; // ownership is transferred from a to b` `} // array is deallocated only once on behalf of its owner, b`  (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Automatic heap deallocation	`fn main() {` `  let a_num = 4;` `  make_and_drop();` `}` `fn make_and_drop() {` `  let a_box = Box::new(5);` `} // a_box gets out of scope, so Rust automatically deallocates 5 from the heap.`  (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
References	References are non-owning pointers that borrows an owning pointer **Example:** `fn main() {` `  let m1 = String::from("Hello");` `  let m2 = String::from("world"); // L1` `  greet(&m1, &m2); // note the ampersands` `  let s = format!("{} {}", m1, m2); // L3` `}` `fn greet(g1: &String, g2: &String) { // note the ampersands`

```
    println!("{} {}!", g1, g2); // L2
}
```



(https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html)

Packages, Crates, Modules

Concept	Snippets/Command/Explanation/Examples
Module tree	

```rust
// lib.rs
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

```
// module tree
crate // src/main.rs and/or src/lib.rs make up the implicit module `crate`
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Absolute vs. Relative path	```rust mod front_of_house { mod hosting { fn add_to_waitlist() {} } } pub fn eat_at_restaurant() { // Absolute path crate::front_of_house::hosting::add_to_waitlist(); // Relative path front_of_house::hosting::add_to_waitlist(); } ```
super (Access items in parent modules)	```rust fn deliver_order() {} mod back_of_house { fn fix_incorrect_order() { cook_order(); super::deliver_order(); } fn cook_order() {} } ```
Making structs and enums public	```rust // all struct fields are private by default pub struct Breakfast { pub toast: String, // public seasonal_fruit: String, // private } // all enum fields are public by default pub enum Appetizer { Soup, // public Salad, // public } ```
use, glob operator	```rust // the `use` keyword brings items from other modules into scope use std::cmp::Ordering; use std::io; // bring into scope multiple items from the same module use std::{cmp::Ordering, io} // nested path using `self` use std::io::{self, Write}; // bring into scope `io` and `io::Write` // bring all public items into scope with the glob `*` operator use std::collections::*; ```

Error Handling	
Concept	**Snippets/Command/Explanation/Examples**

panic!

```
// get a backtrace of a panic!
$ RUST_BACKTRACE=1 cargo run
```

Result enum

```
// `T` is a generic type for the value to be returned in a success case
// `E` is a generic type for the error to be returned in an error case
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Example of recoverable error with `Result`

```
// TOO MUCH `MATCH`, TOO VERBOSE OF AN ERROR HANDLING APPROACH!
use std::fs::File;

fn main() {
    let greetings_file = match File::open("hello.txt") {
        Ok(file) => file,
        Err(error) => panic!("Problem with opening the file: {:?}", error),
    };
}
```

Matching on different errors with `Result`

```
// TOO MUCH `MATCH`, TOO VERBOSE OF AN ERROR HANDLING APPROACH!
use std::fs::File;
use std::io::ErrorKind;

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

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

unwrap, expect

```
// better ALTERNATIVES to `match`
// `unwrap` will return the value if Ok, else call panic! If Err.
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}

// `expect` does what unwrap does, but you get to decide the error message
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

? (Best method)	`// the `?` operator returns the intended return value in a success case` `// and it returns the Error in an error case without calling panic.` `use std::fs::File;` `use std::io::{self, Read};` `fn read_username_from_file() -> Result<String, io::Error> {` `  let mut username = String::new();` `  File::open("hello.txt")?.read_to_string(&mut username)?;` `  Ok(username)` `}`

Generics, Traits, Lifetimes	
Concept	**Snippets/Command/Explanation/Examples**
Function signature with generic types	`fn largest<T>(list: &[T]) -> &T {`
Struct with generic types	`struct Point<T, U> {` `  x: T,` `  y: U,` `}` `fn main() {` `  let both_integer = Point { x: 5, y: 10 };` `  let both_float = Point { x: 1.0, y: 4.0 };` `  let integer_and_float = Point { x: 5, y: 4.0 };` `}`
Enum with generic types	`enum Option<T> {` `  Some(T),` `  None,` `}` `enum Result<T, E> {` `  Ok(T),` `  Err(E),` `}`
Methods with generics	`struct Point<T> {` `  x: T,` `  y: T,` `}` `impl<T> Point<T> {` `  fn x(&self) -> &T {` `  &self.x` `  }` `}`
Traits	`// traits groups method signatures together`

	``rust pub trait Summary { fn summarize(&self) -> String; } pub struct NewsArticle { pub author: String, pub headline: String, pub content: String, } impl Summary for NewsArticle { fn summarize(&self) -> String { format!("{}, by {}", self.headline, self.author) } } pub struct Tweet { pub username: String, pub content: String, pub reply: bool, pub retweet: bool, } impl Summary for Tweet { fn summarize(&self) -> String { format!("{}, by {}", self.username, self.content) } } ```
Trait bounds (impl traits as parameters)	```rust // trait bounds pub fn notify<T: Summary>(item: &T) { println!("Breaking news! {}", item.summarize()); } // syntactic sugar for trait bounds pub fn notify(item: &impl Summary) { println!("Breaking news! {}", item.summarize()); } // another way of writing it with `where` pub fn notify<T>(item: &T) where T: Summary { println!("Breaking news! {}", item.summarize()); } ```
Returning types that share a trait	```rust // Trait `Summary` is implemented on struct `Tweet` fn return_summarizable(username: String, content: String, reply: bool, retweet: bool) -> impl Summary { Tweet { username, ```

	content, reply, retweet, } }
Dangling reference	`// the following code doesn't compile` `// `r` is a dangling reference because it was used after `x` was freed.` `fn main() {` `let r;` `{` `let x = 5;` `r = &x;` `} // x is invalidated` `println!("{}", r);` `}`
Generic lifetime annotation	`// generic lifetime annotation always starts with an apostrophe `'` `// the naming convention is to follow the alphabet ie. 'a, 'b, 'c, ...` `fn main() {` `let result = longest("abcd", "xyz");` `println!("The longest string is {}", result);` `}`  `// the lifetime of the returned value will be equal to the smallest lifetime of the parameters` `fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {` `if x.len() > y.len() {` `x` `} else {` `y` `}` `}`
Struct with lifetime annotations	`struct SomeStruct<'a> {` `part: &'a str,` `}`
‘static lifetime	`// 'static is a special lifetime that means a reference can live for the duration of the program` `let s: &'static str = "I have a static lifetime";`

Making Test Cases	
Concept	**Snippets/Command/Explanation/Examples**
Basic test function structure	`#[cfg(test)]` `mod tests {` `#[test]` `fn test1() {` `let result = 2 + 2;` `assert_eq!(result, 4);` `}`

	`` #[test] fn test2() { let result = 2 + 2; assert_eq!(result, 4); } } ```
assert!	``` // assert! Checks if an expression evaluates to true. If not, it'll fail. #[test] fn test() { // init stuff assert!(/* expression */) } ```
assert_eq!	``` // Check if LHS == RHS #[test] fn test() { let result = 2 + 2; assert_eq!(result, 4); } ```
assert_ne!	``` // Check if LHS != RHS #[test] fn test() { let result = 2 + 2; assert_ne!(result, 5); } ```
Custom error messages	``` #[test] fn test() { let result = 2 + 2; assert_eq!(result, 5, "Test failed! Expected 5, but got {} instead!", result); } ```
[should_panic]	``` #[test] #[should_panic(expected = "Your Error Message")] fn test() { // do stuff that panics } ```
Show program output in test runs	``` $ cargo test -- --show-output ```
Ignore a test function	``` #[test] #[ignore] fn expensive_test() { // this test takes a long time! } ```
Evaluate only some of the test functions	``` // evaluate one test function only $ cargo test <test_function_name> // evaluate all test functions whose names start with a pattern $ cargo test <pattern> ```

	`// evaluate only one test module` `$ cargo test <test_module_name>` `// evaluate only the ignored test functions` `$ cargo test -- --ignored`

Closures, Iterators, Iterator Methods	
Concept	**Snippets/Command/Explanation/Examples**
Closure basic syntax, type inference	`// closures enclose parameters in || instead of ()` `fn main() {` `  let add_one = |x| x + 1; // closures can infer types based on its use` `  println!("The result is: {}", add_one(5));` `}`
Closure can read its environment	`fn main() {` `  let y = 4;` `  let add_y = |x| x + y; // `y` is accessible. Functions can't do this.` `  println!("The result is: {}", add_y(5));` `}`
Iterators	`let v1 = vec![1, 2, 3];` `// iter with immutable reference to each element` `let v1_iter = v1.iter();` `// iter with a mutable reference to each element` `let v1_iter = v1.iter_mut();` `// iter with owned types (dereferenced)` `let v1_iter = v1.into_iter();`
next	`// move on to the next element` `v1_iter.next();`
iter + sum	`// get the sum of an iterator` `let numbers = vec![1, 2, 3];` `let iterator = numbers.iter();` `let sum: i32 = iterator.sum();`
collect	`// `collect` consumes the iterator and gathers the resulting values in a collection` `let chars = vec!['a', 'b', 'c'];` `let some_string: String = chars.into_iter().collect();`
map	`// `map` transforms each element in the iterator. It takes a closure that is applied to each element.` `let numbers = vec![1, 2, 3];` `let squares: Vec<i32> = numbers.iter().map(|&x| x * x).collect();`
filter	`// `filter` returns an iterator over elements that match a predicate.` `let numbers = vec![1, 2, 3, 4, 5];` `let odds: Vec<i32> = numbers.into_iter().filter(|x| x % 2 != 0).collect();`

for_each	`// `for_each` calls a closure on each element of the iterator` `let numbers = vec![1, 2, 3];` `numbers.iter().for_each(|&x| println!("{}", x));`
find	`// `find` searches for an element in the iterator that matches a predicate` `let numbers = vec![1, 2, 3, 4, 5];` `let first_even = numbers.iter().find(|&&x| x % 2 == 0);`
enumerate	`// `enumerate` adds the current count to the iterator's value` `let chars = vec!['a', 'b', 'c'];` `for (index, value) in chars.iter().enumerate() {` `  println!("{}: {}", index, value);` `}`
take, skip	`// `take` takes the first n elements from the iterator` `// `skip` skips the first n elements from the iterator` `let numbers = vec![1, 2, 3, 4, 5];` `let taken: Vec<i32> = numbers.iter().take(3).cloned().collect();` `let skipped: Vec<i32> = numbers.iter().skip(2).cloned().collect();`
all, any	`// `all`, `any` returns a boolean` `// `all` checks if all values satisfies a predicate` `// `any` checks if there exists a value that satisfies a predicate` `let numbers = vec![1, 2, 3];` `let all_positive = numbers.iter().all(|&x| x > 0);` `let any_negative = numbers.iter().any(|&x| x < 0);`
Custom iterator example	`struct Counter {` `  count: u32` `}` `impl Counter {` `  fn new() -> Counter {` `  Counter { count: 0 }` `  }` `}` `impl Iterator for Counter {` `  type Item= u32;` `  fn next(&mut self) -> Option<Self::Item> {` `  if self.count < 5 { // iterate 5 times` `  self.count += 1;` `  Some(self.count)` `  } else {` `  None` `  }` `  }` `}`

Crates, crates.io, Documentation	
Concept	**Snippets/Command/Explanation/Examples**

Dev/Release Optimization	`// Cargo.toml` `// opt-level goes from 0 to 3 (0 means no optimization)` `[profile.dev]` `opt-level = 0` `[profile.release]` `opt-level = 3`
Documentation comments (///)	`// NOTE ANY CODE IN DOCS WILL RUN, SO ASSERT_EQ! WORKS WITH $ cargo test` `/// Adds one to the number given.` `///` `/// # Examples` `///` `/// ```` `/// let arg = 5;` `/// let answer = my_crate::add_one(arg);` `///` `/// assert_eq!(6, answer);` `/// ```` `pub fn add_one(x: i32) -> i32 {` `  x + 1` `}`
Commonly used sections in function docs	Commonly used sections in function docs: - Examples - Panics - Errors - Safety