Variables

Declare with let
`let foo = 2`

Rust has type inference → will assume /detect something is of certain type

Rust has a strong, static type system → comparing vars must be of matching types

sometimes the compiler cannot figure out type

then declare with type `let guess: u32 = 2`

or use turbofish `::<type>` after name of function returning the type

```
let four= "4".parse::<u32>().unwrap()
```

Variables are immutable by default
use mut keyword to declare var as mutable `let mut bar = 5`

Shadowing variables
cleaner conversion between types with same variable name
```
let x = "12"
let x: usize = x.parse().unwrap()
```
or removes need for mut
```
let x = 5
let x = x+1
```
both can use first value when assigning second (or more) time

Ownership of variables

Memory is managed through a system of ownership with a set of rules that the compiler checks at compile time

Ownership rules
1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped

Scope of a variable
within the block it is declared in, which is denoted by brackets

when an owning variable goes out of scoped, it is dropped
```
fn main() {
    let x: i32 = 17;
    {
        let y: i32 = 3;
        println!("x is {} and  y is {}", x, y);
    }
    println!("x is {} and y is {}", x, y); // This won't work.
}
```

Shadowing in scope
shadowing a name does not alter or destroy the value it was bound to
```
let x: i32 = 8;
{
    println!("{}", x); // Prints "8".
    let x = 12;
    println!("{}", x); // Prints "12".
}
println!("{}",x); //prints 8
```

Constants
`const THRESHOLD: i32 = 10;`

type of the value must always be annotated

always immutable

Constants can be declared in any scope, including the global scope
```
const T: usize = 20;
fn main() {
    println!("{}", T);
}
```

Passing move vs copy
the two ways of dealing with assigning or passing value to other variable

Copy: default for variables stored on the stack
Types whose values can be duplicated simply by copying bits

Example of copy
```
let x = 5;
let y = x;
```
both x and y contain a five; each var owns its value because on assigment to y 5 gets copied

what types are copy?
int, float bool, char
tuples containing copy types
types implementing copy trait

A (compound) type can implement Copy if all of its components implement Copy

Move: default for heap variables
"there can be only one"

everything that is not copy is move

examples of move:

assignment
```
let s1 = String::from('hello')
let s2 = s1
println!("{:?}", s1); //ERROR
```
s1 no longer has value, is moved to s2; calling s1 will lead to error

with functions
```
let s = vec![1,2,3,4];
let the_sum = sum_vector(s);
println!("{:?}", s); //ERROR s has moved into sum_vector
```
vars without copy trait are moved to argument if passed to function call!!

with struct and mutability change
```
let mut point = Point { x: 0, y: 0 };
point.x = 5;
let point = point; // now immutable
point.y = 6; // this causes an error
```

Borrowing

By referencing a value, you borrow it instead of move
if you mutably borrow out a value, the owner will no longer be able to change the value, until the reference goes out of scope

borrowing scope does not have ownership so value won't be dropped when borrow goes out of scope

Reference (or borrow) variables with &
immutable by default `&guess` `&mut guess`

a: &T	immutable
mut a: &T	can point p to something else
a: &mut T	can mutate var pointed at by a
mut a: &mut T	can change pointer and var pointed at

https://stackoverflow.com/questions/29672373/what-is-difference-between-mut-a-t-and-a-mut-t

```
let x = &7;
assert_eq!(*x, 7);
let y = &mut 9;
*y = 11;
assert_eq!(*y, 11);
```
```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Rules for referencing / borrowing
you can only have one mutablereference to a particular piece of data in a particular scope → we can use curly brackets to create a new scope, allowing for multiplemutable references, just not simultaneous ones

We also cannot have a mutable reference while we have an immutableone. → multiple immutable references are okay because no one who is just reading the data has the ability to affect anyone else's reading of the data

Automatic dereference
Deref coercion was added to Rust so that programmers writing function and method calls don't need to add as many explicit references and dereferences with & and *.

compiler will insert as many * operations as necessary to get it right
```
f.foo();
(&f).foo();
(&&f).foo();
```
To print a memory address use :p placeholder
```
let x = 1;
println!("x: {}, address: {:p}", &x, &x);
// x: 1, address: #0x7ffc59c6591c
```

Lifetimes
Every reference in Rust has a lifetime, which is the scope for which that reference is valid

lifetimes checked with borrow checker — compares scopes to determine that all borrows are valid

goal: borrowed must not outlive owner → prevents dangling refs

Declaration
lifetime parameters must start with an apostrophe '. `fn from<'a>(text:&'a str)`

function
```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
```
struct/ enum
```
struct Borrowed<'a>(&'a i32)
struct ImportantExcerpt<'a> {
    part: &'a str,
}
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}
```
static lifetime — The 'static lifetime is the entire duration of the program
```
let s: &'static str = "I have a static lifetime."
```

Lifetime elision procedure
when you do not have to declare lifetimes

assign each parameter its own lifetime, and then no need to specify lifetimes when:
1. there is one input parameter. → Its lifetime gets assigned to the output
2. there are multiple parameters, but one is &self or &mut self. → Its lifetime gets assigned to output

if compiler can't figure out lifetime based on these rules, lifetime annotation is needed

Lifetime errors
some simple examples

Lifetime error in scopes
```
let r: &usize;
{
    let x = 5;
    r = &x;
}
println!("r: {}", r);
```
x goes out of scope

so a ref to x is out of scope too

Lifetime error in functions
When we're defining this function, we don't know the concrete values that will be passed into this function, so we don't know whether the if case or the else case will execute. We also don't know the concrete lifetimes of the references that will be passed in, so we can't look at the scopes as w
```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```
```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```
The function signature now says that for some lifetime 'a, the function will get two parameters, both of which are string slices that live at least as long as the lifetime 'a.

Lifetime errors in structs and enums
structs that hold references need to have lifetime annotation
```
struct Holder{
    val: &str
}
let four = String::from("four");
let hold_my_four = Holder{val: four};
```
```
struct Holder<'a>{
    val: &'a str
}
let four = String::from("four");
let hold_my_four = Holder{val: &four};
```

Numbers

Integer
integers are a combination of:
signed (i) or unsigned(u)
8, 16, 32, 64, 128 bits `i128` `u64` etc
```
let x = 2u32  let x: i16 = -8
```
two special cases `isize` `usize`

isize, usize : 32 or 64 dependent on architecture `let x: isize = -12`

floating point
`f32 or f64`

method on numbers
min and max (uint and ints only) — min or max value that can be represented by this integer type. `u8::min_value()` `u8::max_value()`

count bits (uint and ints only) `n.count_ones()` `n.count_zeros()`

to the power `2u32.pow(4)` `2f32.powf(3.0)`

u only `4.is_power_of_two()` `2u8.next_power_of_two()`

sign (i only) `10i8.is_positive()` `10i8.is_negative()` `10i8.signum()` `10i8.abs()`

Checked arithmetic -> Option

ints `16.checked_neg()` `(-32).checked_add(12)` `42.checked_rem(4)` `13.checked_div(3)` `(-25).checked_mul(3)` `32.checked_sub(1)`

u only `checked_next_power_of_two()`

Conversions
string/str <-> digit
```
let  four: u32  =  "4".parse().unwrap()
let  four  =  "4".parse::<u32>().unwrap()
u32::from_str_radix("A", 16) ->Ok(10)
```
char <-> digit
```
'1'.to_digit(10)    -> option<c>
char::from_digit(4, 10)
```

Boolean
`true or false`

true casts to 1 and false to 0

`true as i32` `false as i32`

Boolean operators
and `&` not `!`

or `|` xor `^`

Char
a unicode scalar value — char is always four bytes in size. This is a different representation than a given character would have as part of a String.

indexing into string does not give a char. See string slices

Definition
use single quotes
```
let c = 'a'    let c: char = 'A'    let heart_eyed_cat = '😻'
```

Conversion
num to char `let c = 65u8 as char` `char::from_digit(4, 10)` -> option<char>

char to num `c.to_digit(radix)` -> option<u32>

char to string `c.to_string()` -> String

`"abc".chars()`
```
let mut chars = "abc".chars();
assert_eq!(chars.as_str(), "abc");
assert_eq!(chars.as_str(),"bc" );
assert_eq!(chars.next(), Some('a'));
```
char from byte(s)
```
let sparkle_heart = vec![240, 159, 146, 150];
let sparkle_heart = str::from_utf8(&sparkle_heart)
```
-> option<char>

Pattern matching predicates
```
c.is_digit(radix)        c.is_ascii()
c.is_numeric()           c.is_ascii_digit()
c.is_alphabetic()        c.is_ascii_alphabetic()
c.is_alphanumeric()      c.is_asci_alphanumeric()
c.is_lowercase()  c.is_uppercase()    c.is_ascii_lowercase()  c.is_ascii_uppercase()
c.is_whitespace()        c.is_ascii_whitespace()
                         c.is_ascii_punctuation()
```
c.is_ascii(): this function only recognizes the characters 0-9, a-z and A-Z

Iterators
Returns an iterator that yields the lowercase equivalent of a char as one or more chars.
```
c.to_lowercase() c.to_uppercase()
c.to_ascii_uppercase() c.to_ascii_lowercase()
```
non-ASCII letters are unchanged

`c.len_utf8()` num bytes in utf8 between 1 and 4

Compound types
are derived from more than one primitive type

Tuples
grouping together some number of other values *with avariety of types* into one compound type
```
let tup: (i32, f64, u8) = (500, 6.4, 1)
```
type annotation optional

Access members
You cannot loop / iterate a tuple

dot notation `tup.0` `tup.2`

by destructuring `let (x, y, z) = tup;`

Arrays
fixed length container of some type
```
let a :[type, size]
let a: [i32,10] = [0;10]    let a = [0;10]
let a = [1, 2, 3, 4, 5];
```
type must have copy trait; array with 10 zeros

Access item
see also "Methods for slices and vecs"

`a[idx]` error if out of bounds; `a.get(idx)` ->option; `a.get_mut(idx)`; `array.len()`

Iterate array
```
let arr = [1, 2, 3];
for c in &arr{...}    or    for x in arr.iter() { }
```
arrays are stack allocated; arrays in Rust have a fixed / constant length; this will error
```
let n = 10;
let arr = [i32; n]//error!!
```
see also structs, vecs, hasmap, string for heap allocated compound types

Structs
combine related field and methods into a type

Define struct
Capital case is rust convention

Structs can be nested
```
struct StructName{
    field: type,
    ...
}
struct Rectangle {
    width: u32,
    height: u32,
}
```

Declaration
use mut to make all fields mutable (for private fields, use mods + pub)
```
let user1 = User{
    email:String::from("emailaddress"),
    username:String::from("myusername"),
    [etc]
}
let mut user1 = User{
    email:String::from("emailaddress"),
    username:String::from("myusername"),
    [etc]
}
```

declarating using update pattern ..
.. will copy unspecified field with values from other struct of same type
```
let mut point = Point3d { x: 0, y: 0, z: 0 }
point = Point3d { y: 1, .. point }
```

Destructuring
```
struct Point {   let p = Point{
    x: i32,          x: 2,
    y: i32,          y:12,
}                }
let Point { x: a, y: b } = p;   to declare vars a and b
let Point { x, y } = p;   if var names match field names
```

Define methods
implement by specifying impl block

instance — instance methods take self as first parameter
```
impl StructName{
    fn func(&self, ...){..}
}
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```
three variants, corresponding to the three kinds of things the struct instance could be: &self if it's a reference; &mut self if it's a mutable reference; self if it's a value on the stack.

instance method call: `rect.area()`

static — no self
```
impl StructName{
    fn func(...)...
}
impl Rectangle {
    fn area(width, height) -> u32 {
        width * height
    }
}
```
static method call :: `rect::area(10,3)`

this technique is usually used when the method transforms self into something else and you want to prevent the caller from using the original instance after the transformation

Constructor
In Rust, "constructors" are just a convention:

1. define a struct
```
pub struct Config {
    pub color: Color,
    pub size: Size,
    pub shape: Shape,
}
```
2. set defaults
```
impl Config {
    pub fn new() -> Config {
        Config {
            color: Brown,
            size: Medium,
            shape: Square,
```
3. declare using new or new with update
```
let c = Config {
    color: Red,
    .. Config::new()
};
```
Constructors are static (no self) inherent methods for the type that they construct

Tuple struct
or a tuple with a name

have the added meaning the struct nameprovides, but don't have names associated with their fields
```
struct Color(i32, i32, i32)
struct Point(i32, i32, i32)
let red = Color(100, 0, 0)
let origin = Point(0, 0, 0)
```
extract fields
```
println!("{:?}", red.0); //100
let Color(r,g,b) = red;
println!("{}", r); //100
```

Newtype pattern
distinguish between different interpretations of an underlying type
```
struct Miles(f64);
struct Kilometers(f64);
struct Meters(f64);
```
extract fields
```
let milage = Miles(20);
println!("{:?}", milage.0);
let Mile(m) = milage;
```

Structs are **Move** by default → use derive to implement copy and clone

To enable **print** → use derive to implement Debug

see Traits for hash, defaults, copy
```
#[derive(Debug, Copy, Clone)]
struct Rectangle {
    width: u32,
    height: u32,
}
let r1 = Rectangle{x:1,y:1};
let r2 = r1; //Copied, not moved thanks to copy, clone
println!("copied, not moved {:?}, {:?}", r1, r2);
```

Enums
allow you to define a type by enumerating its possible values

usefull as function parameter: easier to write a function to take multiple types or for vec of type EnumType, which then can contain any type wrapped in the enum → see spreadsheet example in Vec

Definition
```
enum Animal {
    Dog,
    Cat,
}
enum Shape {
    Circle(Point, f64),
    Rectangle(Point, Point),
}
struct Point {
    x: f64,
    y: f64,
}
```

Declaration
```
let mut a: Animal = Animal::Dog    let c = Animal::Cat
```

Enum may have associated data fields "like a tuple"
```
enum Animal {
    Dog(String, f64),
    Cat(String, f64),
}
```
each variant (dog / cat) may have different types
```
let mut a: Animal = Animal::Dog("Cocoa".to_string(), 37.2)
```

Data fields can be named "like a struct"
```
enum Animal {
    Dog(String, f64),
    Cat {name: String, weight: f64 },
}
let c = Animal::Cat { name: "Spotty".to_string(), weight: 2.7 }
```

Enum can have custom discriminant
```
enum Foo {
    Bar,      // 0
    Baz = 123,// 123
    Quux,     // 124
}
let baz_discriminant = Foo::Baz as u32;
assert_eq!(baz_discriminant, 123);
```
you can implements methods on enums, just like you would on structs

the size of an enum will not exceed the largest of its variants plus as much as is needed for the tag

For destructuring enum fields, see match expressions

Strings
"UTF-8 encoded growable string" has ownership over the contents stored on the heap

Declaration
from string literal `String::from("Hello there")`

of certain size `String::with_capacity(int)`

clone `a_string.clone()`

A string literal is a string slice that is statically allocated, meaning that it's saved inside our compiled program, and exists for the entire duration it runs.

Conversion
`str_slice.to_string()` `char.to_string()` `number.to_string()`

string -> bytes (consumes string) `s.into_bytes()` -> Vec<u8>

bytes -> string `String::from_utf8(vec![240, 159, 146, 150])` `String::from_utf8_lossy` will replace any invalid UTF-8 sequences with �

Alteration
inplace; if mutable; string only (no slice)

add string literal	`hello.push_str("..?!")`
add char at end	`hello.push(char)`
insert str	`s.insert_str(index, &str)`
insert char	`s.insert(index,char)` panics if out of index / not on char boundary
remove last char	`s.pop()` option<char>
from index	`s.remove(index)` char (or panics)
split	`s.split_off(index_to_split_at)` inplace = 0..t, returned is t..n
clear	`s.clear()` removes contents, keeps capacity
limit to size [n]	`s.truncate([n])`

String slice &str
second major string type; different from string: it's borrowed

Declaration
from string literal `let hello_world = "Hello, World!";`

from string — whole `string.as_str()` `string.as_mut_str()`

part of `let u = &a_string[i..j]` panics if out of bound; `v.get(i..j)` ->Option(slice); `v.get_mut(i..j)`

Conversions to string
`str_slice.to_string()`

str <-> bytes
`"bors".as_bytes()` -> &[u8] `as_bytes_mut`

`str::from_utf8(&vec![240, 159, 146, 150])`

`from_utf8_mut` -> Result<&str, err>

```
let str_slice: &[&str] = &["one", "two", "three"]
```

Methods for String / &str
P =Pattern char,&str or closure
```
s.rfind(char::is_whitespace)
s.rfind("asd")
s.rfind('A')
```

Transformation: upper/lower/trim
change case -> String `s.to_uppercase()` `s.to_lowercase()` `s.to_ascii_uppercase()` non-ASCII letters are unchanged

remove whitespaces -> &str `s.trim()` `s.trim_start()` `s.trim_end()`

removes matches -> &str `s.trim_matches(P)` `s.trim_left_matches(P)` `s.trim_right_matches(P)`
```
assert_eq!("11foobar11".trim_matches('1'), "foobar");
```

Pattern matching
see also Regex crate

-> bool `s.contains(P)` `s.is_empty()` `s.starts_with(P)` `s.ends_with(P)` `s.is_char_boundary()` `s.is_ascii()`

-> option<index> `s.find(P)` `s.rfind(P)` `s.find(|c:char|(c<'o') && (c>'a'))`

Replacing patterns
-> String

`string.replace(P, &str)`
```
let s = "this is old";
assert_eq!("this is new", s.replace("old", "new"));
```
`s.replacen(P, &str, count)`
```
let s = "foo foo 123 foo";
assert_eq!("new new 123 foo", s.replacen("foo", "new", 2));
assert_eq!("faa fao 123 foo", s.replacen('o', "a", 3));
assert_eq!("foo foo new23 foo", s.replacen(char::is_numeric, "new", 1));
```

Iterate matches
`s.matches(P)` `"1abc2abc3".matches(char::is_numeric).collect()` ->["1", "2", "3"]

`s.repeat(usize)`

Splitting
Iterator -> iterator

chars	`s.chars()`
index + char	`s.char_indices()`
lines	`s.lines()`
words	`s.split_whitespace()`
bytes	`s.bytes()`

```
let mut chars = "abc".chars();
assert_eq!(chars.as_str(), "abc");
assert_eq!(chars.next(), Some('a'));
assert_eq!(chars.as_str(),"bc" );
```
```
let mut bytes = "bors".bytes();
assert_eq!( Some(b'b'), bytes.next());
```

Into two -> slices: at index `let (first, last) = s.split_at(index)`; mutable `s.split_at_mut(3)`

Custom iterators -> iterator

split `s.split(P)` `s.rsplit(P)`

`"abc1def2ghi".split(char::is_numeric)`
```
let t = "abc1def2ghi".split(char::is_numeric);
assert_eq!(vec!["abc", "def", "ghi"], t.collect::<Vec<&str>>());
let u:Vec<&str> = "abc1def2ghi".rsplit(char::is_numeric).collect();
assert_eq!(vec!["ghi", "def", "abc"], u);
```
`"lion::tiger::leopard".split("::").collect()`

splitn — returns at most N items
```
let v: Vec<&str> = "Mary had a little lambda".splitn(3, ' ').collect();
```
->["Mary", "had", "a little lambda"]

Vec<T>
heap allocated containers of *one* type

Declaration
	`let vec : Vec<usize>`
empty	`let mut vec = Vec::new()`
with capacity	`Vec::with_capacity(10)`
with values	`let vec = vec![1, 2, 3, 4]`
from slice/ array	`v = sl.to_vec()` `sl.to_owned()`
multidim	`vec![vec![1], vec![2, 3], vec![4, 5, 6]]`
hold multiple types in vec	see below

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

Add / remove items
Vec only

add item	`vec.push(item)`
insert item	`vec.insert(index, item)`
remove last item	`vec.pop()` ->option
remove from index	`v.remove(index)`
remove and iterate slice	`v.drain(1..).collect()`
append by clone	`vec1.extend(vec2);`
	`vec1.extend_from_slice(&[..])`
append by move	`vec1.append(&mut vec2)`

remove and iterate slice: Creates a draining iterator that removes the specified range in the vector and yields the removed items. append by clone: leaves vec2 as is; append by move: moves vals from vec2

remove all values	`vec.clear()` does not change capacity
shorten vec	`vec.truncate(new_len)` keeps the first len elements and drops the rest
shrink capacity	`vec.shrink_to_fit()`
reserve capacity	`vec.reserve(10)` Reserves capacity for at least additional more elements
Remove consecutive repeated elements	`vec.dedup()`
	`vec.dedup_by(|a, b| a.eq_ignore_ascii_case(b))`
	`vec.dedup_by_key(|i| *i / 10);`
filter/ retain	`v.retain(|a| a<10)` Retains only the elements specified by the predicate

dedup_by: Removes all but the first of consecutive elements in the vector satisfying a given equality relation. dedup_by_key: Removes all but the first of consecutive elements in the vector that resolve to the same key

Array slices
Slices are a view into a block of memory represented as a pointer and a length

Type
```
let x : &[usize] = &[1,2,3]
let x : &mut [usize] = &mut [1,2,3]
```

Vec to slice
```
let s= v.as_slice()      v.as_mut_slice()
let s = &v[1..5]         let s = &mut v[i..n]
```

Slice to array
```
let src = [1, 2, 3, 4];
let mut dst = [0, 0];
dst.clone_from_slice(&src[2..]);
```
`arr.clone_from_slice(&src[2..])` `dst.copy_from_slice(&src[2..])`

Multidim
`&[&v1[..], &v2[..]]`

Methods for slices and vecs

Indexing, changing items
	might panic	option:
get	`v[index]`	`v.get(index)`
change	`vec[index]= 7`	`v.get_mut(index)`

```
let mut arr = [1,2,3,4];
let slice = &mut arr[2..4];
slice[0] = 5;
if let Some(elem) = x.get_mut(1) {
    *elem = 42;
}
if let Some(last) = x.last_mut() {
    *last = 10;
}
```
```
let mut arr = [1,2,13];
let item2 = &mut arr[2];
*item2 = 3;
let item0 = arr.get_mut(0);
*(item0.unwrap()) = 11;
```

other indexing `v.last()` `v.first()` ->option<&T>; `x.first_mut()` `x.last_mut()` ->option<&mut T>

Join /concat
Flatten
```
["hello", "world"].concat()// "helloworld"
[[1, 2], [3, 4]].concat() // [1,2,3,4]
```
with seperator
```
["hello", "world"].join(" ") //"hello world"
[[1, 2], [3, 4]].join(&0) // [1, 2, 0, 3, 4]
```

Change order inplace
`v.sort()` `v.reverse()` (`let x = &mut [11,2,3]; x.sort().reverse();`)

`v.sort_by_key(|k| k.abs())`

`v.sort_by(|a, b| a.cmp(b))`

`v.swap(1, 3)`

`v.swap_remove(1)` vec only — returns item at index, swaps last item to index

Info -> bool
`v.is_empty()` `v.contains(&30)` `v.starts_with(&[10])` `v.starts_with(&[10, 40])` `v.ends_with(&[10])` `v.ends_with(&[10, 40])`

other `vec.capacity()` number of elements the vector can hold; `vec.len()`

Iterate
see iter for maps / filters / plus many others

one at a time
```
for x in &vec{...
for x in &mut vec{..
```
over n elements of the slice at a time

no overlap `slice.chunks(chunksize)` (last slice might be shorter) `v.chunks_mut(2)`
```
for chunk in v.chunks_mut(2) {
    for elem in chunk.iter_mut() {
        *elem += count;
    }
    count += 1;
}
```
overlap `slice.windows(2)`
```
let slice = ['r', 'u', 's', 't'];
let mut iter = slice.windows(2);
```
->ru, us, st, None

Split
into iterator — match not in returned vals
```
let slice = [10, 40, 33, 20];
let mut iter = slice.split(|num| num % 3 == 0);
assert_eq!(iter.next().unwrap(), &[10, 40]);
assert_eq!(iter.next().unwrap(), &[20]);
assert!(iter.next().is_none());
```
```
let mut iter = vec.split(|num|num%3==0)
slice.split_mut(|num| num % 3 == 0)
for group in v.splitn(n, predicate ) {...
for group in v.splitn_mut(n, predicate) {...
```
Returns an iterator over subslices separated by elements that match pred, limited to returning at most n items
```
let v = [10, 40, 30, 20, 60, 50];
for group in v.splitn(2, |num| *num % 3 == 0) {
    println!("{:?}", group);
}
```
(i.e. [10, 40], [20, 60, 50]):

into two
```
let (left, right) = v.split_at(index)
let (left, right) = v.split_at_mut(index)
Some((first, rest)) = x.split_first()
if let Some((last, elements)) = x.split_last()
vec.split_off(usize)
x.split_last_mut()    x.split_first_mut()    option
```
split_at: (slice, slice) 0..index, index..n; split_off: Splits the collection into two at the given index — vec contains 0..usize, returns usize..n

Functions

Declaration
```
fn add_one(x: usize)-> usize{
    x+1
}
```
Type of a function is a function pointer `fn`
```
type some_type = fn(i32, i32) -> i32;
```

Arguments
`(name: type, name: type)`

Types of parameters must be declared

an int
```
fn add_one(x: i32) -> i32 {
    x + 1
}
```
a function
```
fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}
```
tuple `fn measure(measurement:(i32, u32))-> (i32, bool)`

struct `fn create(person: Person)-> Child`

Return type
last expression of function is returned

writing ; after an expression will make it a statement

return early from function by using return keyword

return expressions do need a finishing ;

If there is no expression in the function, the unit type () is returned and must be the return type
```
fn test(a: usize) ->(){
    println!("{}", a)
}
```

Facts about funcs
Functions can be nested — Functions are hoisted

Functions with generic
use traits bounds to make sure that code inside function works on the generic

Syntax
```
fn foo<A, B>(x: A, y: B) {..
fn largest<T>(list: &[T]) -> T {..}
```

With trait bound..
Generics may use traits as bounds on their type parameters — doing so will limit use of func to types that implement trait

..on argument `fn foo<T: Trait>(x: T) {` or `fn foo(x: impl Trait) {`
```
pub fn notify<T: Summarizable>(item: T) {
    println!("Breaking news! {}", item.summary());
}
fn printing(to_print: impl Debug ){
    println!("{:?}", to_print)
}
```
with where syntax `fn foo<T>(x: &[T]) where T: Debug {`

..for multiple trait bounds `<T: Summarizable + Display>`

with where syntax
```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
    U: Clone + Debug{...}
```
..on return type `fn foo() -> impl Trait {..`

are called existential types, return a type that implements some trait(s)

usefull for closures:
```
fn foo() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
fn foo() -> impl Iterator<Item = i32> {
    vec![1, 2, 3]
        .into_iter()
        .map(|x| x + 1)
        .filter(|x| x % 2 == 0)
}
```

Closures
closures can capture their environment and access variables from the scope in which they're defined

Syntax
`let closure = |arg1| {..}` `|par1, par2, ..| for multiple args`

no need for type annotation for params or return value
```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
|| for no args
```

Fn traits
closures can capture values from their environment in three ways

Fn	borrows immutably
FnMut	borrows mutably
FnOnce	takes ownership and can only be called one time

to **force** closure to take ownership use move keyword `let clos = move |z| z==x`

Capturing env with closures
```
fn main() {
    let x = 4;
    let equal_to_x = |z| z == x;
    let y = 4;
    assert!(equal_to_x(y));
}
```

Returning a closure
```
// after rust 1.26 using impl trait
fn foo() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```
```
// before rust 1.26
fn foo() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Storing closures using generics and Fn
```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

Match
allows us to compare a value against a series of patterns and then execute code based on which pattern matches.

Syntax
```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    ...etc...
}
```
`Arm = PATTERN => VALUE`

There must be an arm for every possible value; Pattern must match type and length of value; Arms are scoped locally

Match can assign a value
```
let x = 5;
let number = match x {
    1 => "one",
    5 => "five",
    _ => "something else",
};
```
or do something
```
let x = 1;
match x {
    1 => println!("one"),
    _ => println!("anything"),
};
```

Expressions can:
match a single value	`42 => "some value"`
mutliple values	`0 | 1 | 2  => "not many"`
an enum	`Message::Quit => println!("Quit")`
a range	`1 ... 5 => println!("got 1 to 5")`
anything else	`_ => println!("something else")`

Patterns
named variables
```
let x = Some(5);
match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {:?}", y),
    _ => println!("Default case, x = {:?}", x),
}
```

Guards
an additional if condition for the pattern in an arm
```
Some(x) if x < 10 => process_digit(x)
Some(n) if n == y =>...
```
y scoped from outer scope, n taken from some

Slice patterns
slice patterns can match on size of slice
```
fn foo(s: &[u8]) {
    match s {
        [a, b] => (),
        [a, b, c] => (),
        _ => (),
    }
}
```

Let shorthand
a match that only cares about matching one case; does not check exhaustiveness; only accepts refutable patterns
```
if let Some(3) = somevalue {...      while let Some(top) = stack.pop(){...
```
if let can be combined with (multiple) else branches; or can combine patterns testing different variables
```
if let Some(x) = 1...4{..}
else if(condition){..}...
```
```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();
    if let Some(color) = favorite_color {
        println!("Using {}, as the background", color);
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        ....
    } else {
        println!("Using blue as the background color");
    }
}
```

Binding subpatterns with @
The at operator @ lets us create a variable that holds a value at the same time we're testing that value to see whether it matches a pattern
```
let msg = Message::Hello { id: 5 };
match msg {
    Message::Hello { id: id_variable @ 3...7 } => {
        println!("Found an id in range: {}", id_variable)
    },
    Message::Hello { id: 10...12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```
By specifying id_variable @ before the range 3...7, we're capturing whatever value matched the range while also testing that the value matched the range pattern

The pattern code isn't able to use the value from the id field, because we haven't saved the id value in a variable.

In the last arm, where we've specified a variable without a range, we do have the value available to use in the arm's code in a variable named id. The reason is that we've used the struct field shorthand syntax.

But we haven't applied any test to the value in the id field in this arm, as we did with the first two arms: any value would match this pattern

Ignoring values
`_` underscore as a wildcard pattern that will match any value but not bind to the value

`..` use the .. syntax to use only a few parts and ignore the rest
```
match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    },
}
```
```
fn main() {
    let numbers = (2, 4, 8, 16, 32);
    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}
```

Error Handling and Null values
Wrapping problematic situations into something safe — the enums to prevent problems

the Option enum — Rust explicates situations where value might not exist (be None)

In Rust, in order to have a value that can possiblybe null, you must explicitly opt in by making the type of that value Option<T> .

Expressing null/not null in terms of the type system means the compilercan check that you've handled all the cases you should be handling

the result enum — Rust expicates situations where you do something that might go wrong

used as a return type for function this gives the possibility to distinguish successes from error

By matching the result type you can distinguish and handle a success or error

Both Result and option need to be converted to its type generic to be used

This conversions can only be done by explicating what happens when there is either a none value, or an error

Option<T>
represents an optional value; enum with variants: None, Some<T>

Type
```
let x: Option<usize> = Some(4);    let is_it_a_value = Some(7)
fn somefunc()-> Option<usize>
```

Unpack using match
do something
```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    None => println!("0"),
}
```
assign
```
let n = match opt{
    Some(x) => x,
    None=> 0,
}
```

Checking which enum
->bool `opt.is_none()` `opt.is_some()`

Unwrapping the option
->val if Some(val)..

	..+ panics if None	..+ default when None
panics if None	`opt.unwrap()`	`opt.unwrap_or_else(FnOnce)`
panic with custom_msg	`opt.expect("custom_msg")`	`opt.unwrap_or(default_val)`

```
let k =3;
Some(4).unwrap_or_else(|| 2 * k);
None.unwrap_or("bike")
```

Conversion
replace `opt.take()` replaces value with none, returns new option with value
```
let mut x = Some(2);
let t = x.take();
assert_eq!(x, None);
assert_eq!(t, Some(2));
```
-> to option
```
opt.map(FnOnce)
opt.map_or(defaultVal_if_None, FnOnce)
opt.map_or_else(FnOnce_for_none, FnOnce_for_some)
```
map_or: Applies a function to the contained value (if any), or returns the provided default (if not). map_or_else: Applies a function to the contained value (if any), or computes a default (if not).

-> to result
```
opt.ok_or(err_val)
opt.ok_or_else(FnOnce)
```
mapping Some(v) to Ok(v) and None to Err(err_val); mapping Some(v) to Ok(v) and None to Err(FnOnce())
```
let x = Some("foo");
assert_eq!(x.ok_or(0), Ok("foo"));
let x: Option<&str> = None;
assert_eq!(x.ok_or(0), Err(0));
```

Combining options
`opt1.and(opt2)` returns opt2 if opt1 and opt2 are Some,else none

`opt1.or(opt2)` returns opt1 if some, else opt2

`opt.and_then(FnOnce)` None if opt is None, else calls FnOnce with val in some

`opt.or_else(FnOnce)` Returns the option if it contains a value, otherwise calls f and returns the result.

Result<T,E>
used for returning and propagating errors; enum with the variants Ok(T), Err(E),

Type
```
let x: Result<u32, &str> = Err("Nothing here")
let x: Result<u32, &str> = Ok(12)
fn multiply(num1: &str, num2: &str) -> Result<i32, ParseIntError>
```

Handle result with match
```
let f = File::open("hello.txt");
let f = match f {
    Ok(file) => file,
    Err(error) => {
        panic!(" problem opening file: {:?}", error)
    },
};
```

Checking which enum
-> bool `res.is_ok()` `res.is_err()`

Unwrapping the result
->val if Ok(val)..

	..+ panics if error	..+ return default on error
unwraps ok, else panic	`res.unwrap()`	`res.unwrap_or_else(FnOnce)`
unwraps ok, else panic with custom message	`res.expect("err message")`	`res.unwrap_or_default(default_val)` default value for type

Conversion
with some(val) or None (if error)

To option
```
result.ok()     let x: Result<u32, &str> = Ok(2);  assert_eq!(x.ok(), Some(2));
                let x: Result<u32, &str> = Err("Nothing here");  assert_eq!(x.ok(), None);
result.err()    let x: Result<u32, &str> = Ok(2);  assert_eq!(x.err(), None);
                let x: Result<u32, &str> = Err("Nothing here");  assert_eq!(x.err(), Some("Nothing here"));
```
To res

`res.map(FnOnce)` Maps a Result<T, E> to Result<U, E> by applying a function to a contained Ok value, leaving an Err value untouched.

`res.map_err(FnOnce)` Maps a Result<T, E> to Result<T, F> by applying a function to a contained Err value, leaving an Ok value untouched

Combining results
`res1.and(res2)` res2 if both Ok, else (first encountered) error

`res1.or(res2)` res2 if res1 is err, else Ok val res1

`res.or_else(FnOnce)` Err value if err, else calls F

`res.and_then(FnOnce)` Calls fnOnce if the result is Err, otherwise returns the Ok value of res
```
first_number_str.parse::<i32>().and_then(|first_number| {
    second_number_str.parse::<i32>().map(|second_number| first_number * second_number)
})
```
```
fn sq(x: u32) -> Result<u32, u32> { Ok(x * x) }
fn err(x: u32) -> Result<u32, u32> { Err(x) }
assert_eq!(Ok(2).or_else(sq).or_else(sq), Ok(2));
assert_eq!(Ok(2).or_else(err).or_else(sq), Ok(2));
assert_eq!(Err(3).or_else(sq).or_else(err), Ok(9));
assert_eq!(Err(3).or_else(err).or_else(err), Err(3));
```
```
fn sq(x: u32) -> Result<u32, u32> { Ok(x * x) }
fn err(x: u32) -> Result<u32, u32> { Err(x) }
assert_eq!(Ok(2).and_then(sq).and_then(sq), Ok(16));
assert_eq!(Ok(2).and_then(sq).and_then(err), Err(4));
assert_eq!(Ok(2).and_then(err).and_then(sq), Err(2));
assert_eq!(Err(3).and_then(sq).and_then(sq), Err(3));
```

HashMap<K,V>
`use std::collections::HashMap;`

Declaration
```
let mut map: HashMap<&str, isize> = HashMap::new();
```
using collect on tuples
```
let hash = s.chars().zip(s.chars().rev()).collect::<HashMap<_,_>>();
```
with custom type as key (needs to be hashable)
```
#[derive(Hash, Eq, PartialEq, Debug)]
struct Viking {
    name: String,
    country: String,
}
```
```
let mut vikings = HashMap::new();
vikings.insert(Viking{name:"Einar".to_string(), country:"Norway".to_string()}, 25);
vikings.insert(Viking{name:"Olaf".to_string(), country:"Iceland".to_string()}, 13);
for (viking, health) in &vikings {
    println!("{:?} has {} hp", viking, health);
}
```

Insert /remove
`map.insert(37, "a")` ->option<oldValue> None if key was not yet present

`map.remove(&1)` ->Option<valueAtKey>

`map.retain(|&k, _| k % 2 == 0)` Retains only the elements specified by the predicate.

Entry
entry gets the given key's corresponding entry in the map for in-place manipulation

`Occupied(OccupiedEntry<'a, K, V>)` `Vacant(VacantEntry<'a, K, V>)`

Insert ->&'a mut V
```
map.entry("poneyland").or_insert(12)
map.entry("poneyland").or_insert_with(FnOnce)
map.entry("poneyland").or_default()
let counter = letters.entry(ch).or_insert(0);
*counter += 1;
```
modify ->Self `and_modify`
```
map.entry("poneyland")
   .and_modify(|e| { *e += 1 })
   .or_insert(42);
```

Get/check key/value
`map.get_mut(&1)` `map.get(&1)` Option<&V>

`map[&'a']` panics if no entry

`map.contains_key(&1)` bool

info
`map.len()`

`map.capacity()` Returns the number of elements the map can hold without reallocating

Loop keys /values
order is arbitrary
```
for key in map.keys() {..              ->&'a K
for val in map.values() {              ->&'a V
for val in map.values_mut() {          ->&'a mut V
for (key, val) in map.iter() {         for (key, val) in &map {   ->(&'a K, &'a V)
for (k, val) in map.iter_mut() {       ->(&'a K, &'a mut V)
```

Dates
Chrono provides a DateTime type to represent a date and a time in a timezone. external type https://docs.rs/chrono/0.4.0/chrono/index.html
```
use chrono::prelude::*;
let utc: DateTime<Utc> = Utc::now();            // e.g. `2014-11-28T12:45:59.324310806Z`
let local: DateTime<Local> = Local::now();      // e.g. `2014-11-28T21:45:59.324310806+09:00`
```
```
use chrono::prelude::*;
let dt = Utc.ymd(2017,7,8);   //2014-07-08UTL
let dt = Utc.ymd(2014, 7, 8).and_hms(9, 10, 11);   // `2014-07-08T09:10:11Z`
```

Date / To datetime
`ymd` `ymd_opt`; `and_hms` `and_hms_micro` `and_hms_nano` -> all panic if datetime invalid; `and_hms_opt` -> Option<datetime>
```
let dt = Utc.ymd(2014, 7, 8).and_hms(9, 10, 11); // `2014-07-08T09:10:11Z`
```

Get part of date
```
let dt = Local::now();
dt.year()   dt.month()   dt.day()   start at 1   dt.month0(), day0, ordinal0 for zero start
dt.hour()   dt.minute()  dt.second()             dt.date()  dt.time()
dt.weekday()  -> enum Weekday                    dt.num_days_from_ce()
dt.ordinal ()  -> day of year (1-365)            dt.timestamp()
```
num_days_from_ce: -> number of days since January 1, 1 (Day 1) in the proleptic Gregorian calendar. timestamp: the number of non-leap seconds since January 1, 1970 0:00:00 UTC (aka "UNIX timestamp").

New date with * changed
```
dt.with_year(i32)   dt.with_month(u32)   dt.with_day(u32
dt.with_hour(u32)   dt.with_minute(u32)  dt.with_second(u32)
dt.with_ordinal(32)
use time::Duration;
dt1.signed_Duration_since(dt2)
dt1 + Duration::seconds(num) -> dt     -> duration::seconds
dt.to_string(),  "2014-11-28 12:00:09 UTC"
dt.format("%Y-%m-%d %H:%M:%S").to_string()  "2014-11-28 12:00:09"
```
fn checked_add_signed(self, rhs: OldDuration) -> Option<Date<Tz>>; fn checked_sub_signed(self, rhs: OldDuration) -> Option<Date<Tz>>

Concepts Lingo
Patterns come in two forms: refutable and irrefutable.

Patterns that will match for any possible value passed are irrefutable → let x = 5; because x matches anything and therefore cannot fail to match

Patterns that can fail to match for some possible value are refutable → if let Some(x) = a_value

Function parameters, let statements, and for loops can only accept irrefutable patterns, because the program cannot do anything meaningful when values don't match

Statements vs expressions
Statements are instructions that perform some action and do not return a value. Expressions evaluate to a resulting value

assignment is a statement and does not return a value

Generics
are abstract stand-ins for concrete types or other properties

A way to handle duplication of concepts — Generics allow code to operate on abstract types

When you recognize situations in your code with multiple struct or enum definitions that differ only in the types of the values they hold, you can avoid duplication by using generic types instead

Generics can be used in •function (parameters, return type) •structs field •enums •traits

Syntax
`<T>` signifies a generic in a type declaration; the T is to be substituted for a concrete type; you can use any letter, though starting at T is a rust convention

In functions
```
fn largest<T>(list: &[T]) -> T {..}
```
We read this definition as: the function largest is generic over some type T

In structs
```
struct Point<T> {
    x: T,
    y: T,
}
```
with impl
```
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```
Note that we have to declare T just after impl so we can use it to specify that we're implementing methods on the type Point<T>. By declaring T as a generic type after impl, Rust can identify that the type in the angle brackets in Point is a generic type rather than a concrete type

You can overrule impl<T> with implementation for a specific type
```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```
This code means the type Point<f32> will have a method named distance_from_origin and other instances of Point<T> where T is not of type f32 will not have this method defined

In enums
```
enum Option<T> {    let integer = Some(5);
    Some(T),        let float = Some(5.0);
    None,
}
```

Using multiple generics
```
struct Point<T, U> {
    x: T,
    y: U,
}
let p = Point { x: 5, y: 10.0 }
```
```
fn foo<A, B>(x: A, y: B) {..
```

Traits
A trait tells the Rust compiler about functionality a particular type has and might share with other types

The behavior of a type consists of the methods we can call on that type.

Different types share the same behavior if we can call the same methods on all of those types.

Trait definitions are a way to group method signatures together

Defining a trait
Traits can be implemented for any data type.

declare func signature (summary) that type with trait summarizable needs to implement
```
pub trait Summarizable {
    fn summary(&self) -> String;
}
```
Each type that implements this trait can then provide its own custom behavior

implement on types using impl
```
impl Summarizable for Article{
    fn summary(&self) ->String{..}
}
```
You can also give default implementation for some or all methods of a trait

default overridden by implementing specific for type
```
pub trait Summarizable {
    fn summary(&self) -> String{
        ...default here...
    }
}
```

Associated Types
Associated types are a way of associating a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures.

using an associated type in stead of a generic, results in not having to specify the type when using a trait

when a trait has a generic parameter, it can be implemented for a type multiple times, changing the concrete types of the generic type parameters each time.

When we use a trait, we have to provide type annotations to indicate which implementation of the trait we want to use
```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 {
... }
```
With associated types, we don't need to annotate types because we can't implement a trait on a type multiple times
```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 {
... }
```
```
trait Graph {
    type N;
    type E;
    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
struct Node;
struct Edge;
struct MyGraph;
impl Graph for MyGraph {
    type N = Node;
    type E = Edge;
    fn(.....
}
```

#[derive(trait)]
Rust contains a large number of traits you may use on your types

some operators / functions require a trait to be implemented and will not compile without it

deriving is the simplest way to comply with this trait bound

The following traits can be derived
though not possible in all situations for all types

`PartialEq and Eq`	for equality comparisons
`PartialOrd and Ord`	for ordering comparisons
`Debug`	for programmer output
`Hash`	for mapping a value to a value of fixed size
`Default`	for default values
`Clone and Copy`	for duplicating values

To derive, use:
`#[derive(Trait, Trait2)]`
```
#[derive(Default, Debug)]
struct SomeOptions {
    foo: i32,
    bar: f32,
}
fn main() {
    let options: SomeOptions = Default::default(); //thanks to default
    println!("{:?}", options); //thanks to debug
}
```

Trait bound
limits use of a function to the types implementing the trait
```
fn foo<T: Trait>(x: T) {        fn foo() -> impl Trait {..
or
fn foo(x: impl Trait) {
```

Iter
operations for looping over, mapping, searching, filtering a collection of some sort, or something that yields items

To implement iterator trait define:
Associated type — the type of the thing to iterate over (Associated types connect a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures)

Next function — returns next item or None if there are no more items
```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```
```
struct Counter {
    count: usize,
}
impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
impl Iterator for Counter {
    type Item = usize;
    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    ...
```

Looping through iterator
for method (uses into_iter()) `for c in Counter::new(){...`

Create an iterator
`it.into_iter()`	which iterates over T
`it.iter()`	which iterates over &T
`it.iter_mut()`	which iterates over &mut T

https://stackoverflow.com/questions/34733811/what-is-the-difference-between-iter-and-into-iter — into_iter is a generic method to obtain an iterator, whether this iterator yields values, immutable references or mutable references is context dependent and can sometimes be surprising. iter and iter_mut are ad-hoc methods. This works around the context-dependent bit and, by convention, let you obtain an iterator which will yield references

Next method
`counter.next()` ->Option<item, None>

Iterator to vec
collect — Transforms an iterator into a collection
```
let t: Vec<usize> = Counter::new().into_iter().collect()
let t= Counter::new().into_iter().collect<Vec<usize>>()
```
partition — Consumes an iterator, creating two collections from it.
```
let (even, odd): (Vec<i32>, Vec<i32>) =
    a.into_iter().partition(|&n|n%2==0)
```

Iter provides the following:

Reduction
->bool Tests if every element of the iterator matches a predicate.
```
a.iter().any(|&x| x > 0)   a.iter().all(|&x| x > 0)
```
min/max ->option
```
a.iter().min()  a.iter().max()
a.min_by(|x, y| x.cmp(y))
a.max_by(|x, y| x.cmp(y))
a.min_by_key(|x| x.abs())
a.max_by_key(|x| x.abs())
*a.iter().min_by(|x, y| x.cmp(y)).unwrap(), -10)
```
min_by/max_by: gives the maximum value with respect to the specified comparison function; by_key: max or min after some function has been applied to each element

summary
```
a.iter().product()  a.iter().count()  a.iter().sum()
```

Find item or index ->Option
```
a.iter().find(|&&x| x == 2)
a.iter().position(|&x| x == 2)
a.iter().rposition(|&x| x == 3)
a.iter().nth(1)
a.iter().last()
```
find: Searches for an element of an iterator that satisfies the predicate. position: ->index of first "true". rposition: -> index (from left) of first true, when starting from right. nth: consumes iterator upto n. last: all preceding elements, as well as the returned element, will be consumed from the iterator.
```
let a = [1, 2, 3];
assert_eq!(a.iter().find(|&&x| x == 2), Some(&2));
assert_eq!(a.iter().find(|&&x| x == 5), None);
```
Because find() takes a reference, and many iterators iterate over references, this leads to a possibly confusing situation where the argument is a double reference. You can see this effect in the examples below, with &&x.
```
let arr = vec![1,2,3,4,5,2,7,8,2];
let t = arr.iter().rposition(|&d| *d==2);
assert_eq!(t, Some(8))
```

iterate while holding state
Fold / reduce `let sum = a.iter().fold(0, |acc, &x| acc + x);`

Scan — takes two arguments: an initial value which seeds the internal state, and a closure with two arguments, the first being a mutable reference to the internal state and the second an iterator element. The closure can assign to the internal state to share state between iterations
```
let mut iter = a.iter().scan(1, |state, &x| {
    // each iteration, we'll multiply the state by the element
    *state = *state * x;
    // the value passed on to the next iteration
    Some(*state)
});
```

Adapters
functions which take iterator and return other iterator

map `a.into_iter().map(|x| 2 * x)`

flat_map
```
let words = ["alpha", "beta", "gamma"];
let merged: String = words.iter()
    .flat_map(|s| s.chars())
    .collect();
assert_eq!(merged, "alphabetagamma");
```
enumerate `for (count, val) in a.iter().enumerate()`

chain `let mut iter = a1.iter().chain(a2.iter())`

filter `a.into_iter().filter(|x| x.is_positive())`

filter_map — removes option<t> layer (when compared with filter then map)
```
let a = ["1", "lol", "3", "NaN", "5"];
let mut iter = a.iter().filter_map(|s| s.parse().ok());
assert_eq!(iter.next(), Some(1));
assert_eq!(iter.next(), Some(3));
assert_eq!(iter.next(), Some(5));
assert_eq!(iter.next(), None);
let mut iter = a.iter().map(|s| s.parse()).filter(|s| s.is_ok()).map(|s| s.unwrap());
```
zip
```
let a1 = [1, 2, 3];
let a2 = [4, 5, 6];
let mut iter = a1.iter().zip(a2.iter())
```
unzip
```
let a = [(1, 2), (3, 4)];
let (left,right):(Vec<_>,Vec<_>)= a.iter().cloned().unzip()
//left ==[1,3], right==[2,4]
```

Cycles
Repeats an iterator endlessly

rev	`a.iter().rev();` Reverses an iterator's direction
skip	`a.iter().skip(2);`
skip_while	`a.into_iter().skip_while(|x| x.is_negative());`
take	`a.iter().take(2);`
take_while	`a.into_iter().take_while(|x| x.is_negative());`

```
let mut iter = a.into_iter()
    .skip_while(|x| **x < 0);
// need two *s!
```
Because the closure passed to skip_while() takes a reference, and many iterators iterate over references, this leads to a possibly confusing situation, where the type of the closure is a double reference:

Inspect
for debugging
```
a.iter()
    .cloned()
    .inspect(|x| println!("about to filter: {}", x))
    .filter(|&x| x % 2 == 0)
    .inspect(|x| println!("made it through filter: {}", x))
    .fold(0, |sum, i| sum + i);
```

Env
contains functions to inspect various aspects such as environment variables, process arguments, the current directory, and various other important directories

Get args of main
`for argument in env::args() {...}` An iterator over the arguments of a process, yielding a String value for each argument

Environment vars
get `match env::var(key) {..` Result<String, VarError>

set `env::set_var(key, "VALUE");`

iterate all `for (key, value) in env::vars() {` Returns an iterator of (variable, value) pairs of strings, for all the environment variables of the current process.

Directories
```
pub fn current_dir() -> Result<PathBuf>
pub fn current_exe() -> Result<PathBuf>
pub fn home_dir() -> Option<PathBuf>
match env::home_dir() {
    Some(path) => println!("{}", path.display()),
    None => println!("Impossible to get your home dir!"),
}
```
current_exe: full filesystem path of the current running executable

Path
operations for inspecting a path `use std::path::Path;`

Declare
```
let path = Path::new("./foo/bar.txt");
```
get entries in path `for entry in path.read_dir().expect("failed") {`

->bool
```
Path::new("a_file.txt").is_file()
path.starts_with("/etc")
path.ends_with("passwd")
Path::new("does_not_exist.txt").exists()
Path::new("./is_a_directory/").is_dir()
Path::new("/etc/passwd").has_root()
```

to new ->option
```
Path::new("/etc").join("passwd")
path.with_extension("txt")
path.with_file_name("bar.txt")
Path::new("tmp/foo.txt").file_name()
path.file_stem()        filename without ext
path.extension()
path.display()
```

Inplace methods:
```
buf.set_file_name("bar");
p.set_extension("force");
```

File
A reference to an open file on the filesystem

Open a file in read-only mode `let mut f = File::open("foo.txt")`

Open a file in write-only mode `let mut f = File::create("foo.txt")`

if exist, will overwrite contents of file
```
use std::fs::File;
use std::io::prelude::*;
let mut file = File::create("foo.txt")?;
file.write_all(b"Hello, world!")?;
```
```
use std::fs::File;
use std::io::prelude::*;
let mut file = File::open("foo.txt")?;
let mut contents = String::new();
file.read_to_string(&mut contents)?;
assert_eq!(contents, "Hello, world!");
```
create and open are aliases of openoptions

OpenOptions
poses the ability to configure how a File is opened and what operations are permitted on the open file `use std::fs::OpenOptions;`

read	sets read access
write	sets write access
append	This option, when true, means that writes will append to a file instead of overwriting previous contents.
truncate	If a file is successfully opened with this option set it will truncate the file to 0 length if it already exists.
create	This option indicates whether a new file will be created if the file does not yet already exist.
create_new	No file is allowed to exist at the target location
open	Opens a file at path with the options specified

```
let file = OpenOptions::new()
    .append(true)
    .open("foo.txt");
let file = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("foo.txt");
```

Fs
contains basic methods to manipulate the contents of the local filesystem
```
fs::read_to_string("address.txt")  -> Result<String>
let foo: SocketAddr = fs::read_to_string("address.txt")?.parse()?
fs::copy("foo.txt", "bar.txt")?
fs::create_dir("/some/dir")
fs::create_dir_all("/some/dir")
fs::hard_link("a.txt", "b.txt")
fs::remove_dir("/some/dir")
fs::remove_dir_all("/some/dir")
fs::remove_file("a.txt")
fs::rename("a.txt", "b.txt")
```
copy: copies permissions; returns num bytes copied; will overwrite contents!! create_dir_all: Recursively create a directory and all of its parent components if they are missing. remove_dir_all: Removes a directory at this path, PLUS removing all its contents. rename: overwrites

DirEntry
represents an entry inside of a directory on the filesystem

`for entry in fs::read_dir(dir)` Returns an iterator over the entries within a directory
```
use std::fs;
for entry in fs::read_dir(".")? {
    let dir = entry?;
    println!("{:?}", dir.path());
}
dir.path()      dir.file_name()
dir.file_type()
dir.metadata()  metadata.permissions()
let file = File::open("foo.txt")?;
let mut perms = file.metadata()?.permissions();
perms.set_readonly(true);
file.set_permissions(perms)?;
```

Fmt
Utilities for formatting and printing Strings

to string	`format!`	`format!("{},{}", val1, val2)`
to stdout	`println!` `print!`	`format!("The number is {}", 1);   // => "The number is 1"`
to stderr	`eprintln!`	
to stream	`write!` `writeln!`	see below

```
use std::io::Write;
let mut w = Vec::new();
write!(&mut w, "Hello {}!", "world");
```
placeholders will be filled ..
```
..in order..          format!("{} {}", 1, 2);// => "1 2"
...unless they are...
..named or..          format!("{value}", value=4);// => "4"
..indexed             format!("{name} {}", 1, name = 2)
Can also be intermingled   format!("{1} {} {0} {}", 1, 2); // => "2 1 1 2"
```

Sources
The rust programming language
https://doc.rust-lang.org/book/index.html

Rust by example
https://doc.rust-lang.org/rust-by-example/index.html

The rust standard library
https://doc.rust-lang.org/std/index.html

Link to this map
www.breakdown-notes.com/make/load/rust_cs_canvas/true

Made using Breakdown Notes
www.breakdown-notes.com