


```
println!("x is {}",x);
}
```

```
[E0384] Error: cannot assign to immutable argument `x`
  [command_5:1:1]
7 | fn update(x:i32){
  |             ^
  |             help: consider making this binding mutable: `mut `
8 |     x = x+5;
  |     ^----- cannot assign to immutable argument
  |
  | Note: You can change an existing variable to mutable like: `let mut x = x;`
```

```
In [6]: fn main(){
        let mut y = 77;
        update(y);
        println!("y is {}",y);
        }

        fn update(mut x:i32){
            x = x+5;
            println!("x is {}",x);
        }
```

```
In [7]: main()
```

```
x is 82
y is 77
```

```
Out[7]: ()
```

Cloning

- Makes a copy of a heap value

```
In [8]: let mut s1 = String::from("hello");
        let mut s2 = s1.clone();
        s2.push_str(", world");

        println!("s1 = {}, s2 = {}", s1, s2);
```

```
s1 = hello, s2 = hello, world
```

Transferring ownership via function calls

- Move heap values back and forth
- Here, ownership of `s` moves to function `takes_ownership`

```
In [9]: fn main() {
        let s = String::from("hello"); // s comes into scope

        takes_ownership(s);           // s's value moves into function...
                                        // ... no longer valid here
    } // s goes out of scope. Since s's value was moved, nothing special happens.

    fn takes_ownership(some_string: String) { // some_string comes into scope
        println!("{}", some_string);
    } // some_string goes out of scope, `drop` is called, memory is freed
```

```
In [10]: main()
```

```
hello
```

```
Out[10]: ()
```

If we try to use `s` after its value has moved, we get an error

```
In [ ]: fn main() {
        let s = String::from("hello"); // s comes into scope

        takes_ownership(s);           // s's value moves into function...
                                        // ... no longer valid here

        println!("{}",s);
    } // s goes out of scope. Since s's value was moved, nothing special happens.

    fn takes_ownership(some_string: String) { // some_string comes into scope
        println!("{}", some_string);
    } // some_string goes out of scope, `drop` is called, memory is freed
```


- Transferring ownership requires clumsy mechanisms to "get back" parameters passed to functions

```
In [16]: fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String
    (s, length)
}
```

```
In [17]: main()
```

The length of 'hello' is 5.

```
Out[17]: ()
```

References

- Point to a variable that contains a value on the heap
- Avoids moving ownership
- Creating a reference results in *borrowing* the value

```
In [18]: fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

```
In [19]: main()
```

The length of 'hello' is 5.

```
Out[19]: ()
```

- Arguments passed as references are not automatically mutable
- Use `&mut` to denote a mutable reference

```
In [ ]: fn main() {
    let s = String::from("hello");
    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
[E0596] Error: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
└─[command_20:1:1]
6   fn change(some_string: &String) {
7       some_string.push_str(", world");
    └─ help: consider changing this to be a mutable reference: `mut`
    └─ `some_string` is a `&` reference, so it cannot be borrowed as mutable
Note: You can change an existing variable to mutable like: `let mut x = x;`
```

```
In [21]: fn main() {
    let mut s = String::from("hello");
    change(&mut s);
    println!("s is {}",s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

```
In [22]: main()
```

s is hello, world

```
Out[22]: ()
```

Constraints on mutable references

- One mutable reference is permitted

```
In [23]: {
  let mut s = String::from("hello");
  let r1 = &mut s;
  println!("{}", r1);
  println!("{}", s);
}
```

hello
hello

Out[23]: ()

- But now only the reference can update the value, not the original variable

```
In [ ]: {
  let mut s = String::from("hello");
  let r1 = &mut s;
  r1.push_str(", world");
  s.push_str(" again");
  println!("{}", r1);
  println!("{}", s);
}
```

[E0499] Error: cannot borrow `s` as mutable more than once at a time

```
[command_24:1:1]
3   let r1 = &mut s;
      └── first mutable borrow occurs here
5   s.push_str(" again");
      └── second mutable borrow occurs here
6   println!("{}", r1);
      └── first borrow later used here
```

- Cannot have two or more mutable references
- Avoids *race conditions* in concurrent programs

```
In [ ]: {
  let mut s = String::from("hello");

  let r1 = &mut s;
  let r2 = &mut s;

  println!("{}", r1, r2);
}
```

[E0499] Error: cannot borrow `s` as mutable more than once at a time

```
[command_25:1:1]
4   let r1 = &mut s;
      └── first mutable borrow occurs here
5   let r2 = &mut s;
      └── second mutable borrow occurs here
7   println!("{}", r1, r2);
      └── first borrow later used here
```

- Here the second mutable reference is created after the first one goes out of scope, so this is fine

```
In [26]: {
  let mut s = String::from("hello");

  {
    let r1 = &mut s;
  } // r1 goes out of scope here, so we can make a new reference with no problems.

  let r2 = &mut s;
}
```

Out[26]: ()

- Cannot mix immutable and mutable references

- Again to avoid race conditions

```
In [ ]: {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{}", r1, r2, r3);
}
```

[E0502] Error: cannot borrow `s` as mutable because it is also borrowed as immutable

```
[command_27:1:1]
4   let r1 = &s; // no problem
      |
      | immutable borrow occurs here
6   let r3 = &mut s; // BIG PROBLEM
      |
      | mutable borrow occurs here
8   println!("{}", r1, r2, r3);
      |
      | immutable borrow later used here
```

- Here the last use of `r1` and `r2` occurs before `r3` is declared
- Rust does sophisticated static analysis to determine this at compile time

```
In [28]: {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{}", r1, r2);
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem
    println!("{}", r3);
}
```

```
hello and hello
hello
```

Out[28]: ()

- Unlike `gives_ownership` earlier, here `dangle` returns a reference
- Potential problem --- when `dangle` exits, `s` goes out of scope and `reference_to_nothing` becomes a *dangling pointer*, pointing to nothing
- Rust catches this as a compile-time error

```
In [ ]: fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

[E0106] Error: missing lifetime specifier

```
[command_29:1:1]
5   fn dangle() -> &String {
      |
      | expected named lifetime parameter
      | help: instead, you are more likely to want to return an owned value: ``
      | help: consider using the ``static`` lifetime, but this is uncommon unless you're returning a
      | borrowed value from a `const` or a `static`: ``static``
```

Slices

- A function to compute the length of the first word in a string
- `bytes.iter()` iterates through `bytes`, `enumerate()` returns a pair (index,reference to value), which is decomposed through pattern matching into `(i, &item)`

- `b' '` specifies a byte constant for the space character

```
In [30]: fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

- In this function, Rust cannot recognize that the return value is an index into the string
- If we clear the string, the index is no longer valid, but cannot be flagged by compiler

```
In [31]: fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}
```

- A string slice is written similar to a slice in Python
- Gives a reference to a substring

```
In [32]: {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```

Out[32]: ()

- Rewrite `first_word` to return slice corresponding to first word
- Will examine distinction between `&String` and `&str` later

```
In [33]: fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

- Now, if we try to clear the "parent" string while holding a reference to a substring, it is a compile error
- Another example of combining immutable and mutable references --- the call `s.clear()` implicitly passes a mutable reference to `s` to `clear()`, while `word` currently holds an immutable reference

```
In [ ]: fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

[E0502] Error: cannot borrow `s` as mutable because it is also borrowed as immutable

[command_34:1:1]

4 let word = first_word(&s);

immutable borrow occurs here

6 s.clear(); // error!

mutable borrow occurs here

8 println!("the first word is: {}", word);

immutable borrow later used here