# Lesson 04

Iterators

## Iterators

Rust, like python, has iterators. In Rust, iterators are just entities which implement the *Iterator* trait. The iterator trait has a lot of methods, however almost all of them have default implementations. In fact, the only method that you have to implement in order to create an iterator is the *next* method. The relevant part of the trait looks like this:

### Iterator Trait

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>
}
```

An iterator simply returns Some(value) when invoking next() until it is empty, at which point it returns a None. Iterators are used all over the place. In fact, you have been using them for some time.

Notice that this implies the need to store state. It would be inconvenient and sometimes not possible to implement this directly on anything you want to iterate over. Hence, we often create an iterator type that is distinct from the type we iterating over. So how do we get an iterator for a given type? Enter the *IntoIterator* trait. Now this definition is a bit tricky, so lets take a second to soak it in:

Here is the trait:

### IntoIterator Trait

```
pub trait IntoIterator
where
    <Self::IntoIter as Iterator>::Item == Self::Item,
{
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

Notice that into_iter takes a "self", not a &self. That means it consumes self when called on non references.

What does this all mean? Lets develop an intuition by looking at Vec in the standard library. And specifically the IntoIterator signature. First, we will find that there isn't just one implementation. While Rust does not have function overloading, it does have trait overloading.

### For Items

```
impl<T> IntoIterator for Vec<T> {
  type Item = T //The type of the elements being iterated over.
  type IntoIter = IntoIter<T> //Which kind of iterator are we turning this
into?

  fn into_iter(self) -> IntoIter<T>
}
```

### For References

```
impl<'a, T> IntoIterator for &'a Vec<T> {
  type Item = &'a T //The type of the elements being iterated over.
  type IntoIter = Iter<'a, T> //Which kind of iterator are we turning this
into?

  fn into_iter(self) -> Iter<'a, T>
}
```

### For Mutable References

```
impl<'a, T> IntoIterator for &'a mut Vec<T> {
  type Item = &'a mut T //The type of the elements being iterated over.
  type IntoIter = IterMut<'a, T> //Which kind of iterator are we turning
this into?

  fn into_iter(self) -> IterMut<'a, T>
}
```

So what is IntoIter? Lets take a look. Ahh. It is implemented for Vec in the std::vec namespace. So std::vec::IntoIter. And IntoIter implements the Iterator trait. Aha, so we have a bespoke struct for vectors that is the iterator.

Here is the signature of the impl:

```
impl<T> Iterator for IntoIter<T> {
  type Item = T; //The type of the elements being iterated over.
  fn next(&mut self) -> Option<T> {...}
}
```

So this is interesting. Here we have an impl for all types T. A generic implementation. Which is good, because we wouldn't want to have to implement Iterator for every possible type that a Vec can hold.

What about usage in practice? Lets see:

```
let myvec = vec!["a", "b", "c"];
// here we use &myvec because we dont want to consume it
for v in &myvec.into_iter() {
  println!("{}", v);
}
```

Once we have an iterator, we have access to a ton of trait methods with default implementations. For the python savvy, everything in `itertools` and more is at your disposal. These methods are for the most part *iterator adapters*. An iterator adapter is a function on an iterator which returns another iterator. Here are some common adapters from the *Iterator* trait:

- map - returns an iterator which applies a function to each item
- filter - returns an iterator which applies a predicate and keep those items that pass
- enumerate returns an iterator which gives the current iteration count as well as the next value as a (cnt, val) pair.
- flat_map - returns an iterator which flattens a nested iterable
- fold - this is the classic reduce.
- zip - an iterator that iterates over two other iterators

## For Loops - Sugar

One thing to know about rust - for loops are just syntactic sugar for iterators. Any struct which implements IntoIter may be used in a for loop.

```
let values = vec![1, 2, 3, 4, 5];
for x in values {
  println!("{}", x);
}
// desugars into

  let result = match IntoIterator::into_iter(values) {
      mut iter => loop {
          let next;
          match iter.next() {
              Some(val) => next = val,
              None => break,
          };
          let x = next;
          let () = { println!("{}", x); };
      },
  };
```

iterators are lazy. Something needs to drive them. You can use a for loop, but you dint have to. Iterator has a method called *collect*, which can take anything iterable, and turn it into a relevant collection.

The most basic pattern in which collect() is used is to turn one collection into another. You take a collection, call into_iter on it, do a bunch of transformations, and then collect() at the end.

One of the keys to collect()'s power is that many things you might not think of as 'collections' actually are. For example, a String is a collection of chars. And a collection of Result<T, E> can be thought of as single Result<Collection, E>. See the examples below for more.

Because collect() is so general, it can cause problems with type inference. As such, collect() is one of the few times you'll see the syntax affectionately known as the 'turbofish': ::<>. This helps the inference algorithm understand specifically which collection you're trying to collect into.

```
fn main() {
    let  a = vec![1, 2, 3];
    // if we didnt use (&a) then we would consume a here and not be able
to
    // call trippled
    let doubled = (&a).into_iter().map(|x| x * 2).collect::<Vec<_>>();
    println!("{:?}", doubled);

    // here we let the type inference system figure out what type Vec
collects
    let trippled: Vec<_> = (&a).into_iter().map(|x| x * 3).collect();
    println!("{:?}", trippled);
}
```

rust playground

## String vs &str

Rust provides two distinct types to deal with strings: A String is an owned representation of a string, very similar to a std::string in c++. It is grow-able, and largely heap allocated under the hood. For example:

```
let mut s = String::from("Hello Rust");
s.push_str(". I forgot the period");
println!("{} has this length {}", s, s.len())
```

Specifically, a String stores three scalars on the stack:

1. a pointer to a head allocated u8 buffer
2. a capacity tracking the total allocated size of the heap buffer
3. and a length, which may be different.

(of course you can box up a string, and then it is all on the head except for the box pointer, but lets not get into that.)

A &str on the other hand is a *string slice*. It is an immutable view of a string. A &str is basically a pointer to the backing data along with knowledge of the size. For C programmers:

```
struct str {
  char text[]; // unknown number of bytes
};

// like a fat pointer
struct &str {
  size_t length;
  const char *text; // not owned
};

struct String {
  size_t length;
  size_t free capacity;
  char *text; //owned
};

oh and a &String is a *String;
```

It should be noted that str and &str are fundamental data types, whereas String is a struct in the standard library. A str and thus a &str can thus have its own methods, whereas a &String is just a reference to a String. You will rarely if ever deal with &String. However, you will deal with &str all the time.

If you want to pass a string without handing over ownership use &str. String has a nice convenience method called `as_str()` which will cast it to a &str. Equivalently, you can just put an '&' in front of the string.

**Into**

One other convenience trait will be important for our exercise. There are a pair of traits: Into and Form which are particularly useful for bounding generic args. It is often the case that you want to accept a String or a &str, and convert to a String in either case. You can do this with the Into trait, which is part of the standard prelude, meaning it will be automatically in scope.

```
pub struct Person {
    name: String
}

impl Person {
    /// new up a bob
    pub fn new<I>(name: I) -> Self
    where
        I: Into<String>
    {
        Self {
            name: name.into()
        }
    }
```

```
    /// say hi
    pub fn hi(&self) {
        println!("Hi,my name is {}", &self.name);
    }
}

fn main() {
    // you can do this
    let p1 = Person::new("Bob"); // &'static str
    let otherbob = String::from("OtherBob");
    let p2 = Person::new(otherbob.as_str()); // reference. Additional
allocation
    let p3 = Person::new(otherbob);// move otherbob into Person
    p1.hi();
    p2.hi();
    p3.hi();
}
```

rust playground

For more on &str vs String, see this blog post