2_Iterators_en

May 2, 2023

1 Iterators

The iterator pattern allows you to perform some task on a sequence of items in turn. An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished.

In Rust, iterators are lazy, meaning they have no effect until you call methods that consume the iterator to use it up.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter(); // creating an iterator
}
main();
```

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter { // use the iterator
        println!("Got: {}", val);
    }
}
main();
```

The **next** method returns one item of the iterator at a time wrapped in *Some* and, when iteration is over, returns *None*.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();
    match v1_iter.next() {
        Some(&value) => println!("Vector element: {:?}", value),
        None => (),
    }
}
main();
```

Also note that the values we get from the calls to **next** are immutable references to the values in the vector. The **iter** method produces an iterator over immutable references.

If we want to create an iterator that takes ownership of v1 and returns owned values, we can call **into_iter** instead of **iter**.

Similarly, if we want to iterate over mutable references, we can call **iter_mut** instead of **iter**.

Methods that call **next** are called consuming adaptors, because calling them uses up the iterator. One example is the **sum** method, which takes ownership of the iterator and iterates through the items by repeatedly calling **next**.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum(); // type annotation is mandatory

    println!("Total: {}", total);
}
main();
```

`Iterator::any` is a function which when passed an iterator, will return `true` if any element satisfies the predicate. Otherwise `false`.

[ ]:
```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` for vecs yields `&i32`
    println!("2 in vec1: {}", vec1.iter().any(|&x| x == 2));
    // `into_iter()` for vecs yields `i32`
    println!("2 in vec2: {}", vec2.into_iter().any(|x| x == 2));
}
main();
```

`Iterator::find` is a function which when passed an iterator, will return the first element which satisfies the predicate as an `Option`.

[ ]:
```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` for vecs yields `&i32`.
    let mut iter = vec1.iter();
    // `into_iter()` for vecs yields `i32`.
    let mut into_iter = vec2.into_iter();

    // A reference to what is yielded is `&&i32`. Destructure to `i32`.
    println!("Find 2 in vec1: {:?}", iter.find(|&&x| x == 2));
    // A reference to what is yielded is `&i32`. Destructure to `i32`.
    println!("Find 2 in vec2: {:?}", into_iter.find(|&x| x == 2));
}
main();
```

1.1 Methods that produce other iterators

Other methods defined on the Iterator trait, known as **iterator adaptors**, allow you to change iterators into different kinds of iterators. You can chain multiple calls to iterator adaptors to perform complex actions in a readable way. But because all iterators are lazy, you have to call one of the consuming adaptor methods to get results from calls to iterator adaptors.

The iterator adaptor method *map*

[ ]:
```
let v1: Vec<i32> = vec![1, 2, 3];

let v: Vec<_> = v1.iter().map(|x| x + 1).collect();
```

- The `map` method produces another iterator for the result of the function passed to it, applied on the original elements yielded by the iterator.
- The `collect` method consumes the iterator and collects the resulting values into a vector.

This vector will end up containing each item from the original vector incremented by 1.

`Iterator::enumerate` creates an iterator which gives the current iteration count as well as the next value.

The iterator returned yields pairs `(i, val)`, where `i` is the current index of iteration and `val` is the value returned by the iterator. https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.enumerate

[ ]:
```
fn main() {
    let a = ['a', 'b', 'c'];

    for (index, element) in a.iter().enumerate() {
        println!("Element at position {} is {}", index, element);
    }
}
main();
```

2 Exercises

Exercise 1: Write a function that returns a vector of `GeoCoordinate` from a vector of latitudes and a vector of longitudes.

[ ]:
```
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}


// TO DO (1) Write a function that returns a vector of GeoCoordinate coordinates
// write a version with enumerate, and another with zip and map
// try to write versions in imperative and functional styles

fn main() {
    // TO DO (2) Test your implementation

}
main();
```

Solution:

```
[ ]: #[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

fn imperative(latitudes: &[f64], longitudes: &[f64]) -> Vec<GeoCoordinate> {
    let mut points: Vec<GeoCoordinate> = Vec::new();
    for (index, element) in latitudes.iter().enumerate() {
        points.push(GeoCoordinate{latitude: *element, longitude:␣
 ↪longitudes[index]});
    }
    points
}

fn functiona_bad(latitudes: &[f64], longitudes: &[f64]) -> Vec<GeoCoordinate> {
    latitudes
        .iter()
        .enumerate()
        .map(|(index, latitude)| GeoCoordinate {
            latitude: *latitude,
            longitude: longitudes[index]
        })
        .collect()
}
```

```
fn functional_idiomatic(latitudes: &[f64], longitudes: &[f64]) ->␣
 ↪Vec<GeoCoordinate> {
    latitudes
        .iter()
        .copied()
        .zip(longitudes.iter().copied())
        .map(|(latitude, longitude)| GeoCoordinate {
            latitude,
            longitude
        })
        .collect()
}

fn main() {
    let latitudes = vec![1.0, 3.0, 5.0];
    let longitudes = vec![2.0, 4.0, 6.0];
    let result = functional_idiomatic(&latitudes, &longitudes);
    println!("{:#?}", result);
}
main();
```