Iterators

Learn to Code with Rust / Section Review

Iteration

- To **iterate** means to perform an action repeatedly.
- Iteration in Rust means repeating the same operation on a sequence of items, one item at a time.
- We can perform manual iteration with constructs like **loop** and **while** but the idiomatic way is to use the iterator tools built into Rust.

The **Iterator** Trait

- The **Iterator** trait indicates that a type is iterable. It can be traversed over one element at a time.
- The **Iterator** trait mandates a **next** method that returns an **Option**.
- The **Some** variant will store a value of the next iterator element.
- The **None** variant indicates that the iterator has been exhausted.
- Once we implement **next**, we get access to 75+ helper methods that depend on its implementation.

```
pub trait Iterator {
    type Item;

    // Required method
    fn next(&mut self) -> Option<Self::Item>;
```

The **IntoIterator** Trait

- The **IntoIterator** trait indicates that a type can be converted into an iterator.
- The **IntoIterator** trait mandates a **into_iter** method that returns an iterator.
- The **Item** associated type represents the type of the iterator's yielded elements.
- The **IntoIter** associated type declares the iterator type that the **into_iter** method will return.

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    // Required method
    fn into_iter(self) -> Self::IntoIter;
}
```

The **FromIterator** Trait

- The **FromIterator** trait indicates that a type can be constructed *from* an iterator.
- The **FromIterator** trait mandates a **from_iter** associated function that returns the type that the trait is being implemented for.
- The **FromIterator** trait requires a generic representing the type of the iterator's yielded elements.
- The **collect** method calls the **from_iter** function behind the scenes.

```
pub trait FromIterator<A>: Sized {
    // Required method
    fn from_iter<T>(iter: T) -> Self
       where T: IntoIterator<Item = A>;
}
```

The **for** Loop

- The **for** loop creates and iterates over an iterator.
  - ```
    for element in collection {
        // Logic
    }
    ```
- The **for** loop declaratively exhausts the iterator. There is less room for error.
- Provide an iterator variable name. The iterator will assign each yielded iterator element to the variable.
- The iterator variable becomes unavailable after iteration.
- Within the block, declare the logic to perform on each yielded element.
    ```

Iterator Construction Methods

- The **into_iter** method returns an iterator that yields the elements by value (transferring ownership if elements do not implement the **Copy** trait).
  - for value in **collection.into_iter()**
  - for value in **collection**
- The **iter** method returns an iterator that yields immutable references to the elements.
  - for value in **collection.iter()**
  - for value in **&collection**
- The **iter_mut** method returns an iterator that yields mutable references to the elements.
  - for value in **collection.iter_mut()**
  - for value in **&mut collection**

Yielded Elements

- Iterators from different types will yield different elements.
- An array or vector iterator will yield each element from the collection.
- A **HashMap** iterator will yield a tuple of a key and its value.
- A **String** iterator can yield characters (the **chars** method), bytes (the **bytes** method), even individual lines (the **lines** method).

Lazy Iterators

- Iterators are lazy. They will not exhaust/consume themselves until an operation or method forces them to iterate.
- A **for loop** or the complementary **for_each** method force an iteration.
- The **collect** method gathers the iterator's elements into a collection type.

Consuming Methods

- The **any** method returns true if any iterator element satisfies a condition.
- The **all** method returns true if all iterator elements satisfy a condition.
- The **partition** method chunks/buckets the elements that satisfy and do not satisfy a condition. It returns a tuple with the two groups.

Mathematical Methods

- The **sum** method returns the sum of the iterator's values. The **product** method returns the product.
- The **max** and **min** methods return an **Option** with the largest or smallest value from the iterator.
- The **count** method exhausts an iterator by counting its elements.

Positional Methods

- The **last** method returns an **Option** with the last iterator element.
- The **nth** method extracts an element from a specific index position within the iterator.
- The **nth_back** method extracts an iterator element proceeding *backwards*.
- The **position** method returns an **Option** with the index position of the first iterator element that satisfies a condition.

The **fold** and **reduce** Methods

- The **fold** method exhausts an iterator to build up and produce a single value.
- The **fold** method receives a starting value and a closure as an arguments.
- Each iteration will receive the accumulator (the data that persists over the iterator) and the current element.
- The **reduce** method is similar but it supplies the first iterator element as the starting accumulator value.

The **sort** and **sort_by_key** Methods

- The **sort** method mutates an array or vector to sort its elements in ascending order (smallest to largest, alphabetical).
- The **sort_by_key** method can sort an array of structs. The closure declares which struct field to use for the sort.
- The **is_sorted** method returns true if the collection is sorted in ascending order.

The **reverse** Method

- The **reverse** method reverses the order of elements in the collection.
- Applied after a sort, the **reverse** method creates a descending sort (largest to smallest, reverse alphabetical).
- The **reverse** method is called directly upon the collection. Don't confuse it with the **rev** method on an iterator.

Intro to Adapter Methods

- An **adapter method** is one that transforms an iterator into another one by performing a logical operation.
- The performance is lazy. Nothing is computed until the final iterator is exhausted/consumed.
- We can chain adapter methods to create a *pipeline* or *sequence* of transformations to apply.

Common Adapter Methods

- The **map** method performs an operation on each iterator element to produce an iterator of new values.
- The **filter** method selects for iterator elements that satisfy a predicate condition.
- The **filter_map** method performs both a filter and a map in sequence. Return a **Some** variant to keep an element, return a **None** variant to exclude it.

More Adapter Methods

- The **enumerate** method creates an iterator that yields a tuple with the index position and the element (in that order).
- The **zip** method merges two iterators together and yields a tuple with the elements at each shared index position.
- The **copied** method returns an iterator that creates a copy of each element. Elements must implement the **Copy** trait.
- The **cloned** method returns an iterator that calls **clone** on every element. Elements must implement the **Clone** trait.

Even More Adapter Methods

- The **flatten** method transforms an iterator of collections into a single iterator of flattened values.
- The **skip** method bypasses a specified number of elements from the iterator.
- The **take** method limits the iterator by selecting a number of elements from its start.
- The **step_by** method produces an iterator that yields values at specified intervals/steps/sequences.
- The **rev** method reverses the order of elements in an iterator.

Collecting Command Line Arguments I

- The **env::args** function returns an **Args** struct that stores command line arguments. The struct implements the **Iterator** trait.
- The first yielded element will be the executable file's name.
- Bypass the first element transforming the iterator with **skip(1)**.
- Use the **take** method to limit the iterator to the first **n** elements.

Collecting Command Line Arguments II

- Pass command line arguments after double dashes (**--**).
  - **cargo run -- kings and queens**
- The syntax ensures the arguments flow into our program rather than the **cargo** command.
- Separate command line arguments with a space.
- Command line arguments will arrive in the program as Strings.

Reading Directory I

- The **fs::read_dir** function returns a **Result<ReadDir>**.
- The **ReadDir** struct implements the **Iterator** trait and yields **Result<DirEntry>** elements.
- A **DirEntry** is a struct representing either a file or a folder. Its **path** method returns the location/path of the entity.

Reading Directory II

- The **fs::metadata** function accepts a path and returns a **Result<Metadata>**.
- The **Metadata** struct includes methods like **is_file** and **is_dir** to figure out the type of a directory entry.
- There are multiple idiomatic approaches to handle **Result** values: the **match** keyword, **if let** construct, the **unwrap_or** method, and the try operator (**?**).
- The try operator unwraps the **Ok** variant's value or returns early, propagating the **Err** variant's data upwards.