Iterators

Intro

- This is a concept used in many programming languages but specializes in the Functional programming like closures
- Iterators encapsulates the logic for iterating over a collection of any type (arrays, vectors, graphs, etc) and lets the user simplify the process for all collections
- This allows us to iterate over numerous data structures in a uniform war

Using Iterators

- Iterators are lazy be default
- Iterators don't do anything until they are used
- Example:

  ```
  let v1 = vec![1, 2, 3];

  let v1_iter: Iter<i32> = v1.iter();
  ```

  - Iterators is a specific type or object from a collection
  - Its a type `Iter<T>` for collection like `Vec<T>`
- Example use case:

  ```
  let v1 = vec![1, 2, 3];

  let v1_iter = v1.iter();

  for val in v1_iter {
      println!("Got: {}", val);
  }
  ```

  - Notice that in the for loop there is no logic used for retrieving the value of each element because that logic is encapsulated in `Iter`
  - For instance, there is no `get()` method or `get_value()` method or `get_key_value_pair()`

Iterator Trait

- All iterators and collections that use iterators need to implment the `pub trait Iterator` which looks like the following:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

- This has a new syntax here `type` this in its own can be a chapter but a brief summary of `type`
    - This keyword is meant to be used like the following:
    - `type Meters = f64;`
    - In other words you make a synonym/alias for `f64` and use `Meters`.
    - This does not replace the name of `f64` but instead allows a new name for the same type
    - To showcase its functionality:

    ```
    type Kilometers = i32;
    let x: i32 = 5;
    let y: Kilometers = 5;
    println!("x + y = {}", x + y);
    ```

    - When used in a `trait`, classes that implement that trait needs to specify what alias will be tied to
    - For example: `type Item = T;` in this case it is a generic type
    - The main purpose of using the keyword `type` is to make a complex type into a single name

    ```
    type Thunk = Box<dyn Fn() + Send + 'static>;

    let f: Thunk = Box::new(|| println!("hi"));
    ```

        - We simplified here the `Box<dyn ....>` type into `Thunk`
- Looking back into the example with `Iterator` trait
- There is a `type` that needs to be implemented and a function `fn next(&mut self)-> Option<Self::Item>`
- Note that the function returns an Options which if possible will return a `Some(Item)` value

- To see how the `next function` works with a iterator, we can make a test in rust

- Example:

```
#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();
```

```
    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

- Here we are testing the value we get from the iterator against what we expect to find
- Notice we are expecting `Some(&type)`
- This means that `vector` objects when they implement trait `Iterator` it assigns `type Item = &T;`
- Whenever we iterate over a collection its best practice to iterate over an immutable borrowed reference so we do not take ownership of the item itself
- We can make it borrow a mutable reference by using `vec.iter_mut()` instead of `vec.iter()`
- We can also make it take ownership by using `vec.into_iter()` and this returns `owned` values

Iterator Built in Default Methods

- We know that iterator is a trait and needs two aspects impemented but there are several default implemented methods in this trait
- These methods are put into two categories **Consumers** and **Adaptors**
- Consumers are methods that take in the iterator and return a integer or float or some other data type
  - Consumers traverse through the collection and collects some data and returns the data
- Adaptors take in the iterator and return an iterator
  - Adaptors also traverse through the collection but it performs an operation on each element and returns a new iterator for the collection's new data
- You probabaly have seen this behavior in functional programming and how Java or Python adopts functional programming
- Consumer Example:

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

  - Here the iterator calls `sum()` which takes in `&self` as a parameter and calls `next()` internally
  - Each time it calls `next()` it sums with the previous values and returns the result after summing all the values
- Adaptor example:

```rust
let v1: Vec<i32> = vec![1, 2, 3];
v1.iter().map(|x| x + 1);
```

- You might recognize `map()` if you ever worked with functional programming languages
- If not, this function takes in two parameters a function (or closure in rust) and an iterator using `&self`
- It will apply the closure on each element as the iterator calls `next()` and returns an iterator to the beginning of the new collection's elements
- In our case we will get a warning because we do not utilize the return iterator
- In other words, iterator adaptors are **lazy** and do not perform an **inplace operation**
- This is why we need to add another function to **consume** the iterator into a collection using `collect()`

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

  let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

  assert_eq!(v2, vec![2, 3, 4]);
  ```

  - Here `collect()` collects the resulting values from applying the closure into a vector
- Adaptors are a little odd as they require a Consumer method at all times to collect the result of the adaptor method

Understanding the Power of Closures in Adaptors

- Since Closures captures their environment variables, it is easy and quick to make simple closure functions to place inside an Adaptor
- Lets take a look at the example below:

```rust
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
```

```
    fn filters_by_size() {
        let shoes = vec![
            Shoe {
                size: 10,
                style: String::from("sneaker"),
            },
            Shoe {
                size: 13,
                style: String::from("sandal"),
            },
            Shoe {
                size: 10,
                style: String::from("boot"),
            },
        ];

        let in_my_size = shoes_in_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
    }
}
```

- Without overcomplicating this example
- We can see that a method `fn shoes_in_size()` which filters a vector of our Shoe struct
- Now `filter()` is a common adaptor in functional programming that takes in a closure which returns a `bool` and this way it can filter each element out of a collection and return a new collection with only the elements that you want
- We can see that the closure takes in `s` which represents each element in the collection
- It also uses `shoe_size` which is a variable within the same scope of the closure
- This allows us to compare the elements in the collection easily against captured enviornment variables in a closure

Making our own Iterator

- We will make a new Iterator by first making a Struct and that struct will implement the trait `Iterator`
- Example:

```rust
struct Counter{
    count: u32,
}

impl Counter{
    fn new(count: u32) -> Counter{
        Counter { count }
    }
}

impl Iterator for Counter{
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        }
        else{
            None
        }
    }
}
```

- Brief description here, we made a `Counter` struct that has one field called `count`
- It implements a new method
- It also implements `Iterator` trait
- We defined the `type` as `u32`
- We also implemented the `next()` method
  - This meethod will mutate the value by incrementing by 1, and then return an Option
  - However it will stop iterating after 5 iterations
  - This is just to make a failing case

- We can now look at how this is used in a test suite

```rust
#[test]
fn calling_next_test(){
    let mut counter = Counter::new(1);

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

  - In this test we see that everytime we call `next()` on the counter object, we increment the counter

- Once it hits 5, then we return `None`
- Let's see how this looks by using the Adaptor methods with this iterator

```
#[test]
fn using_defualt_iterator_trait_methods(){
    let sum: u32 = Counter::new(0 as u32)
        .zip(Counter::new(0 as u32).skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();

    assert_eq!(sum, 18);
}
```

  - This is a very complicated example just to show how different adapter methods can work together
  - So here we make a new `Counter` and call multiple methods,
  - We first call `zip(self, Iterator)`, this takes in two iterators including itself and combines them into one iterator of `pairs` and `values` from both iterators.
  - Then we use `skip(i32)` on the second parameter of `zip`. `skip()` is an adaptor which skips the **first n** elements
  - Outside of `zip` we call the `map` adaptor. This time we use a closure that takes in a tuple of pairs. Remember that zip returns an iterator of a collection of pairs. Map now will return an iterator where each element is the product of the pairs
  - Of the products collection we use another adaptor `filter`. Here we are filtering with any product that is divisible by 3. i.e. we will only have numbers that are a multiple of 3 in the collection
  - Lastly we call the consumer `sum()` to gather the data of the filtered products and sum them together