Iterators | Implementing ***IntoIterator***
Using a Custom Iterator

Mini Iterator Review

- ***Iterator*** trait allows iteration over a collection
  - Yield items
  - Struct must be mutable & contain iteration state information
- ***IntoIterator*** trait defines a proxy struct & determines how data is accessed
  - Move, borrow, mutation

Problem

- Implementing ***IntoIterator*** allows control of the iteration, *but...*
  - We aren't using an existing collection to store data
    - No ***.iter()*** or ***.into_iter()***
  - We don't want to pollute our data structure with iteration information

Solution

- Make an intermediary struct
  - Implement ***Iterator***
    - Mutable, handles iteration state
- Implement ***IntoIterator*** on data struct
  - Combined with the intermediary struct will allow iteration

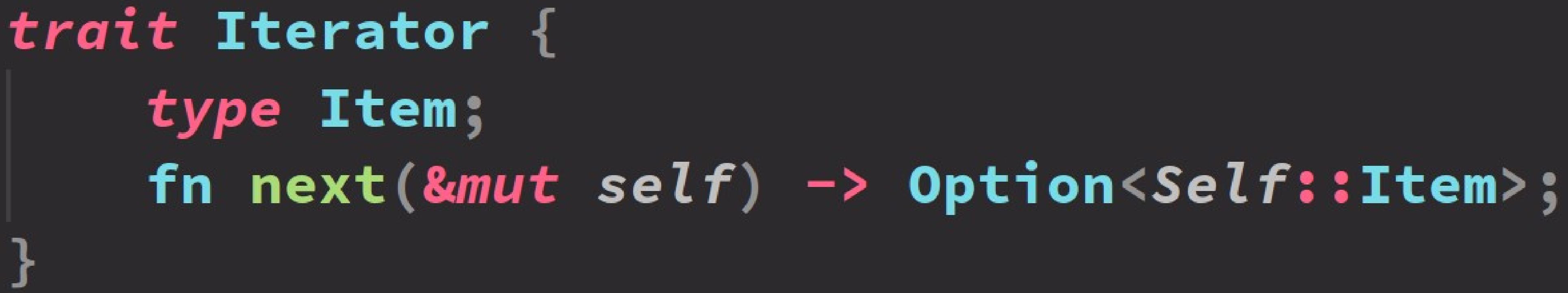
Setup

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct ColorIntoIter {
    color: Color,
    position: u8,
}

struct ColorIter<'a> {
    color: &'a Color,
    position: u8,
}
```

Review – Iterator Trait



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

Impl Iterator - Move

```rust
impl Iterator for ColorIntoIter {
    type Item = u8;
    fn next(&mut self) -> Option<Self::Item> {
        let next = match self.position {
            0 => Some(self.color.r),
            1 => Some(self.color.g),
            2 => Some(self.color.b),
            _ => None,
        };
        self.position += 1;
        next
    }
}
```

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct ColorIntoIter {
    color: Color,
    position: u8,
}
```

Impl IntoIterator - Move

```rust
impl IntoIterator for Color {
    type Item = u8;
    type IntoIter = ColorIntoIter;

    fn into_iter(self) -> Self::IntoIter {
        Self::IntoIter {
            color: self,
            position: 0,
        }
    }
}
```

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct ColorIntoIter {
    color: Color,
    position: u8,
}
```

Done!

```
let color = Color {
    r: 10,
    g: 20,
    b: 30,
};
for c in color {
    println!("{}", c);
}
```

```
10
20
30
```

Overview

```rust
let color = Color {
    r: 10,
    g: 20,
    b: 30,
};
for c in color {
    println!("{}", c);
}
```

```rust
struct ColorIntoIter {
    color: Color,
    position: u8,
}
```

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

Impl Iterator - Borrow

```rust
impl<'a> Iterator for ColorIter<'a> {
    type Item = u8;
    fn next(&mut self) -> Option<Self::Item> {
        let next = match self.position {
            0 => Some(self.color.r),
            1 => Some(self.color.g),
            2 => Some(self.color.b),
            _ => None,
        };
        self.position += 1;
        next
    }
}
```

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct ColorIter<'a> {
    color: &'a Color,
    position: u8,
}
```

Impl IntoIterator - Borrow

```rust
impl<'a> IntoIterator for &'a Color {
    type Item = u8;
    type IntoIter = ColorIter<'a>;

    fn into_iter(self) -> Self::IntoIter {
        Self::IntoIter {
            color: &self,
            position: 0,
        }
    }
}
```

```rust
struct Color {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct ColorIter<'a> {
    color: &'a Color,
    position: u8,
}
```

Done!

```rust
let color = Color {
    r: 10,
    g: 20,
    b: 30,
};
for c in &color {
    println!("{}", c);
}
for c in &color {
    println!("{}", c);
}
```

```
10
20
30
10
20
30
```

Notes

- Non-trivial to implement mutable iteration using ***IntoIterator***
  - Collect mutable references into a Vector and return it
  - Use ***unsafe*** to bypass compiler checks
- Prefer using existing ***.iter()*** methods on structures when possible
  - Vectors, HashMaps, etc
  - Easier to work with, covers most cases

Recap

- Custom iteration requires a dedicated iteration struct for each type of data handling mechanism
  - Move, borrow
- Prefer using the ***.iter()*** methods on existing collections if possible