COMP 333
Summer 2025

Enums and Pattern Matching in Rust

1.a.) Define an `enum` named `MyBool` which represents truth and falsehood. You can name the cases whatever you want *except* for `true` and `false`, since `true` and `false` are already reserved words in Rust.

1.b.) Define a `to_bool` method on your `enum`, which will take a reference to a `MyBool` instance, and return a `bool` representing the value as a `bool`.

1.c.) Define a `print_value` method on your `enum`, which will print either `true` or `false`, depending on the value of the enum. The method should not need to take ownership over the `MyBool` instance.

2.a.) Define an `enum` named `StringOrBool`, which has two cases: one holding a `String`, and another holding a `MyBool` instance. Example usage is below:

```
let ex1: StringOrBool =
  StringOrBool::StringCase("foo".to_string());
let ex2: StringOrBool =
  StringOrBool::BoolCase(MyBool::False);
```

2.b.) Define a `to_bool` method for `StringOrBool`, which takes a reference to a `StringOrBool` instance, and will return a `bool`. `to_bool` should do the following:

- If the `StringOrBool` instance is a `StringCase`, then it should return `true` if the length of the string is greater than 10, else `false`. You can check the length of a `String` using the `len()` method, defined on `String`.
- If the `StringOrBool` instance is a `BoolCase`, then it should return whatever the `to_bool` method for `MyBool` returns (which you defined in 1.b.). You should call `MyBool`'s `to_bool` method instead of redefining it.