**Error Handling Funtime**

Oh noes, the clever TAs at App Academy made this "super useful" library, but it keeps throwing ugly error messages that are hard to understand. Let's revamp the library to throw more descriptive errors and prevent incorrect usage.

**Learning Goals**

* Know how to raise and rescue an exception
* Be able to explain how an exception bubbles up after it is raised
* Know when to use ensure and retry
* Be able to choose an appropriate exception class

**Phase 1: Setup**

Download the project [skeleton](https://assets.aaonline.io/fullstack/ruby/projects/error_handling_funtime/skeleton.zip). You will primarily be working in super\_useful.rb to improve errors in our library. The user's script, aptly named user\_script.rb, will be using the functions and classes defined in super\_useful.rb.

**Phase 2: Make friendly monster (maybe) let you try again**

**Overview**

Sometimes when an error is thrown we would like to try the failing operation again (hopefully with different input :wink:). This is often the case with user input and text parsing. Let's try to make friendly monster happy by allowing us to retry feeding it a fruit when certain errors are thrown.

**Instructions**

Friendly monster is *really* friendly and *really* likes coffee, so he'd like to give us another try, but only when we give him "coffee".

First, handle the errors being thrown by #reaction in #feed\_me\_a\_fruit.

Note that #reaction throws errors receiving an argument that is not in FRUITS. Next, let's differentiate the errors thrown so our calling function, #feed\_me\_a\_fruit, can try to feed friendly monster again, but only when we have given it coffee.

Now that we have different error types being thrown by #reaction we can do a little conditional logic in #feed\_me\_a\_fruit to retry the failing block of code again, but only if it is a coffee-related error.

**Recap**

Being able to rescue and retry failing code gives us even more control over the flow of our program. Handling different errors separately gives us even more control.

**Phase 3: Ensure**BestFriend**is a real best friend**

**Overview**

Another use case for raising errors is to enforce correct usage of code. For example, if a function requires its arguments to be of certain types in order to execute properly, it might be best to check their type before executing any logic. This is useful because it allows us to inform the user that they are not using our function properly instead of raising a runtime error, which may seem like a bug in our code or be more difficult to debug.

**Instructions**

If we look at user\_script.rb, we see that our dear user thinks you can be besties if you've known each other less than a year. We do not agree. Friendships, like a fine wine, need at least five years to mature. Update BestFriend#initialize in super\_useful.rb to raise a descriptive error when yrs\_known is less than 5.

Test your code, then assume the role of our dear user and update our call to BestFriend#new to create a *real* friendship (yrs\_known >= 5).

Our dear user also thinks it's okay to leave name and fav\_pastime empty when creating a new instance of BestFriend. But it's not okay. It leaves #do\_friendstuff and #give\_friendship\_bracelet sorely lacking. Poorly formatted text just makes us seethe with displeasure. Update the initialize method to raise descriptive errors when given strings of length <= 0.

Test your code, then again assume the role of our dear user and update our call to BestFriend#new.

**Recap**

Raising errors for invalid arguments can ensure that our code is used the way we intend. However, be aware that the types of inputs we can receive are numerous. We don't want or need to be checking against every possible type for each argument we receive.

**Resources**

* Exceptions, error handling reading in sidebar
* [Skorks on exceptions](http://www.skorks.com/2009/09/ruby-exceptions-and-exception-handling/)
* [Ruby Patterns](https://github.com/adomokos/DesignPatterns-Ruby/)