←Prev Submission Next→

4 Ruby: Debugging Code

Even the Best Programmers Get Bitten

Nobody writes perfect code. It is important to be able to read errors that the interpreter provides and to know where to look for the solution.

> Some terminology: When something goes wrong in code, the interpreter will **"raise"** an **"error"**. If you predict that error ahead of time, you can determine how that error is **"handled"**, by **"rescuing"** it and then specifying behavior to follow.

When a program doesn't run as expected, consider the following process:

1. Read the error message carefully. Errors raised by the Ruby interpreter are generally descriptive and will point you in the right direction to fix the error.
2. While the error message raised may tell you the line where the error occurred, the solution may sometimes involve changing multiple lines of code. In addition to reading the line of code the error message references, read the entire method to ensure you have the proper context.

Error Types

If you defined the `hello` method as:

```
def hello(name)
  "Hello #{name}"
end
```

...and then tried to call the `hello` method like this:

```
hello("George", "Washington")
```

You'd receive an error message stating:

```
ArgumentError: wrong number of arguments (2 for 1)
```

In this example, the error message told you exactly what the problem was. The message is explaining that the method was passed two arguments instead of one. If you call the method in a Ruby interpreter, a line number is provided. After receiving the message you can read the line of code to see what method was called. You can then assess the method's definition and fix the arguments.

Debugging Thought Process

Some methods behave incorrectly without raising errors. Try another example by writing a method that takes a birthday date as an argument. The method should return the day of the week that the birthday will fall on in the coming year.

```
require 'date' # Makes Ruby's date library available to our code

def next_birthday(birthday)
  a_year_off = birthday.next_year
  a_year_off.strftime("%D")
end

next_birthday(Date.new(2013, 6, 1))
```

This method returns: `"06/01/14"`, so it returned the next year's date, the first of June 2014, but not the day of the week. Where did we go wrong? Break this method down:

1. We defined a `next_birthday` method which takes one argument - `birthday`.
2. Within the method, we assigned the value of `birthday.next_year` to a variable called `a_year_off`.
3. We called `strftime("%D")` on `a_year_off`.

You might be tempted to read about `next_year` to figure out what's going wrong, but first, let's narrow down the problem.

The method finds the right date, but returns a string representing the whole date, not the word representing the month. So we can actually assume that `next_year` is working correctly, and our problem is likely with the *next* line. The **docs for `next_year`** confirm this. We're assigning the correct value to `a_year_off`, but failing to convert that value to the desired string.

So we've narrowed the problem down to the `strftime` method.

> It's important to narrow problems down logically before digging too deep.

Now that we're relatively confident what's wrong, let's review the **Ruby Docs on the `strftime` method**, which we suspect we're using incorrectly. The method takes a "format string" argument and formats the time it is called on (here `a_year_off`) using that string.

Look through the docs for the "%D" format string; it formats time exactly as we received it.

> What output would we expect if we called `a_year_off.strftime('%Y')`?

We want the day of the week, which means we need to pass in a different formatting string. Which string do we need?

This example introduced a new object: `Date`. Mostly, however, it introduced a method of debugging: we use the information we've received to create a hypothesis ("We're using `strftime` incorrectly on the right date.") and then test it. This is how you should debug your code. The more you practice, the faster you will read code and figure out where to implement a fix.

Exercises

✓ ~~No Method Error~~

✓ ~~Wrong Number of Arguments~~

✓ ~~No Name Error~~

✓ ~~Type Error~~

✓ ~~Unexpected End~~

Solution Walk-Throughs

Do not watch these videos until after you've attempted your own solutions. If you struggle to complete these exercises, submit your best effort to your mentor *before watching a solution walk-through*. Submit your answers even if some tests don't pass, as it's important for your mentor to understand your thought process. Discuss difficult exercises with your mentor.

There are many possible solutions to the exercises. A walk-through is meant to provide insight into the thought process of *one possible solution* per exercise. If your solution is different yet passes all the tests, do not change your solution to match what is seen in the video. Instead, discuss your thought process with your mentor.

1. **No Method Error Solution**
2. **Wrong Number of Arguments Solution**
3. **No Name Error Solution**
4. **Type Error Solution**
5. **Unexpected End Solution**

assignment completed

?