RPN Calculator

This RPN calcualator takes strings, and returns floating point numbers which correspond to the evaluation of the string.

Give the test cases a spin! They produce these results:

- "1 2 +" => 3
- "4 2 /" => 2
- "2 3 4 + *" => 14
- "3 4 + 5 6 + *" => 77
- "13 4 -" => 9

If the string is invalid input, then the calculator returns a suitable error. Give these test cases a spin as well:

- "1 +" => Exception: "incorrect number of arguments"
- "a b +" => Exception: "invalid number"
- "1 2" => Exception: "incorrect number of arguments"

Code explanation

Here is the program, in literate Ruby:

```
def rpn(str)
    tokens = str.split(" ")
```

The string is split in to tokens. So, if the string were "1 3 +", then `tokens' would be ["1", "3", "+"] here.

```
    res = tokens.reduce([]) do |stack, next_token|
        ...
    end
```

We'll chew through (`reduce') the token array, one at a time. In the semantics of `reduce', the first argument is the accumulator (the `stack'), which represents what we've chewed through already (it starts empty: `[]'). The second argument is token that we're currently chewing. So, in the first example ("1 3 +"), during the last step of `reduce', the `stack' would be [1,3], while `next_token' would be "+".

Let's look at the code inside the reduce.

Depending on the `next_token', we'll do different things.

```
case next_token
    ...
end
```

- If it's an operation ("+" or "-" or "*" or "/"), we'll:
    - pop the two last seen tokens off the stack
        - if one of them is nil, it means that at least one pop failed, so we raise an exception
    - combine the two popped tokens with the operation, keeping in mind the order (1 - 3 != 3 - 1)
    - push the result onto the stack again

```
when "+", "-", "*", "/"
    first, second = stack.pop(2)
    if !first or !second then raise "incorrect number of arguments" end
    result = first.send(next_token, second)
    stack << result
```

- If it's a string with at least one number inside, and contains only numbers
    - we convert it to a number (float, actually, in this case), and push it on to the stack

```
when /^\d+$/
    stack << next_token.to_num
```

- Otherwise, it shouldn't belong in our string. So, we raise an exception, noting that the input was valid.

```
else
    raise "invalid input"
```

At the end, `res' contains the result of our chewing and reducing. If our input string was valid, we should only have one number in our stack: the result. If so, we'll just reutn that. If, however, there are multiple items in the stack, that means that there were more numbers than symbols, so some numbers couldn't be reduced. In that case, the input string is considered malformed, so we raise an exception, stating that there is an "incorrect number of arguments".

```
res.count == 1 ? res.first : raise("incorrect number of arguments")
```

The `to_num' method is also worth a quick look. Essentially, it takes a string ("123"), converts it to an integer (123), then to a float (123.0). Why we do the floating point conversion is explained in the next section.

It begins by considering the string character by character, then chews through it, this time manually (without a `reduce') (also explained in a design decision). We start with `accum' = 0. If the character's ascii value is not within the ascii range for numbers (48 > val or 57 < val), the value of the whole string is nil. Otherwise, we update our `accum' by multiplying it by 10

(since we're in decimal system), then add our number. At the end, we return the `accum'.

```
def to_num
    accum = 0.0
    for char in self.each_char
        ascii_val = char.ord
        if ascii_val < 48 || ascii_val > 57
            return nil
        else
            int_val = ascii_val - 48
            accum *= 10
            accum += int_val
        end
    end
    accum
end
```

Notably, the `to_num' method is meta-programmed to be a part of the String class. The rationale for this is also explained below.

Design decisions

- **Why use `reduce'?**
    - As a lover of Haskell, folding from the left seems like the best way to reduce a list of values to a single one. Arguably, to a Ruby beginner / colleagues, this isn't the clearest code. Alternatively, I could have written it iteratively, whilst declaring a stack variable beforehand. That's shown below.
    - Since my recruiter told me to come up wih a "novel" solution, using a `reduce' seemed like a reasonable idea.
    - If I were pushing to production, it's quite likely I'd use the iterative version instead (since most developers would be more familiar with it.)
    - (Bonus: it shows off my knowledge of Ruby and of functional programming.)

```
def rpn(str)
    stack = []
    for token in str.split(" ")
        case token
        ...
```

- **Why use `send'?**
    - It's significantly cleaner than having another case statement branch off of `next_token'. Alternatively, it could have been written like that (shown below), but it adds too much verbosity, without much improvement in clarity. (Commenting the `send' explains its purpose well, I think.)
    - We could also match on the individual operators (instead of matching on all 4), which would save us having to nest a if inside of a match. However, since there's no easy construct like `before_filter' (from Rails controllers), we'd have to push

and pop in every single case (also shown below). That's ridiculously verbose, so I haven't used it.

Nesting ifs inside of a case:

```
case "+", "-", "*", "/"
    first, second = stack.pop(2)
    if next_token == "+"
        result = first + second
    elsif next_token == "*"
        result = first * second
    ...
```

Using a separate case statement:

```
case "+"
    first, second = stack.pop(2)
    stack.push(first + second)
case "*"
    first, second = stack.pop(2)
    stack.push(first * second)
...
```

- **Why does `to_num' return an int disguised as a float, instead of an int?**
  - If it returned an int, we might divide 2 by 4, which results in a 0, since int / int = int. While a situation like that wasn't stated in the problem description, it seemed like a pretty obvious drawback. In any case, I instead return a float, so 2.0 / 4.0 = 0.5.
  - Errors should be obvious to the user. If the user input is valid, the program should respond gracefully. Producing "1" when given "3 2 /" is un-graceful, and the error is non-obvious to the user. So, my options were to either throw an exception, or add floating point. Adding floating point seemed like the better option.

- **Why is `to_num' a method inside the String class?**
  - It makes it easier to call, since calling "123".to_num looks better than doing atoi("123"). It's more Ruby-ish, essentially.
  - (Bonus: it shows off my knowledge of Ruby and meta-programming.)

- **Why does `to_num' not use a `reduce'?**
  - Mainly because I need to return a nil if the string contains something other than a number. Returning from inside a block is sketchy.
  - (Bonus: it shows off my knowledge of Ruby and of meta-programming.)

- **Why hard-code in the values for ascii integers?**
  - The alternative is to put them as constants, but I don't think that's much better, in this case. If I were pushing to production, I might consider it, but it seems like

it mostly adds verbosity without much clarity gained.

- **Why raise "incorrect number of arguments" instead of "not enough arguments"?**
  - It's ambiguous whether "1 3 + +" is actually supposed to mean "1 3 +" or "1 3 + 42 +". So, the error should be as general as possible.

- **Why is `calculator.rpn' a separate file? Why isn't it inside `rpn.rb'?**
  - This has to do with testing. Since I had to be able to call the calculator from the command line, it needed to take arguments. Since I needed to require the file (for testing), and since require doesn't take arguments, I had to refactor the code s.t. it could be tested as well as called. That way, I could call `rpn.rb', while testing `calculator.rb'.

- **Why throw specific types of exceptions?**
  - Easier to test, since testing the message of exceptions isn't generally good form. (I do it anyways, though, since I've written the code all by myself.)

- **Why not use inline rescues?**
  - It turns out you can't catch specific types of exceptions when you inline rescue (http://stackoverflow.com/questions/19924795/using-single-line-conditional-with-require-rescue) .