Monday February 4

Project 2 Review
  featuring more examples and random Ruby topics...

Project Submissions
  - Send individual files
  - Send sources (plain text files) so I can load and run them
  - Your documentation should explain what you did when you wrote the program
  - One or two paragraphs is usually sufficient
  - Topics to consider:
    - what were the main challenges?
    - is there anything you want me to notice?
    - what would you do to extend the program?
    - how did you test the program?

Grading Project 2
  - I did not put letter grades or numeric scores on these projects
  - I wrote short notes on the documentation and in the programs themselves
    - look for lines that start with # (or in some cases @)
  - I tried to give hints on how to fix any errors I found
  - I also tried to answer any questions that were raised in the documentation
  - Send me mail or come see me during office hours if you still have questions

Code Samples
  - The following slides have little pieces of Ruby code
    - some from programs turned in by students
    - some from my programs
  - I don’t want to give the entire programs for each project (yet)
    - I want to encourage people to go back and modify their own programs
    - use as much of the code from these slides as you want
    - let these examples inspire some experiments or extensions
The assignment was to modify this program, which sums elements from the command line:

```ruby
sum = 0
ARGV.each do |n|
  sum += n.to_i
end
puts sum
```

The new program should also print the mean, which is the sum divided by the number of elements

- The number of elements is just the length of ARGV

Aside: ARGV

- Many glitches with this project (and others) can be traced to issues with ARGV
- Here is a little program that explores ARGV in a command line program:

```ruby
#!/usr/bin/env ruby
# What is ARGV?
puts "ARGV is a #{ARGV.class}"
puts "it has #{ARGV.length} elements"
ARGV.each_with_index do |x, i|
  puts "ARGV[#{i}] = #{x} (a #{x.class})"
end
```

Sample output:

```
% args.rb foo 0 3.14159
ARGV is a Array
it has 3 elements
ARGV[0] = foo (a String)
ARGV[1] = 0 (a String)
ARGV[2] = 3.14159 (a String)
```

- A new method: `a.each_with_index` is like `a.each`, but it has two arguments, the array element and its position

Bottom line:

- ARGV is an array of strings
- If your app needs to assign a meaning to items you need to convert them
- Use `to_i`, `to_f`, `File.open()`, ...

Back to sum.rb

- The easy way to compute the mean: divide `sum` by `ARGV.length`
  - It’s also OK to define another counter `n`, and add 1 to `n` each time through the loop
- A problem with simply dividing by `ARGV.length`: the result is an integer
- Here is an irb experiment to learn about integer arithmetic:

```ruby
>> sum = 19
19
>> n = 4
4
>> sum / n
4
```

- A simple way to get the mean to be a float: convert the sum to a float before doing the division, e.g.

```ruby
>> sum.to_f / n
4.75
```
grep.rb

- The goal for this program was to search for a specified pattern in the input stream
- Example:
  
  ```
  % grep.rb puts sum.rb
  puts "ARGV is a #{ARGV.class}"
  puts "it has #{ARGV.length} elements"
  puts "ARGV[#{i}] = #{ARGV[i]} (a #{ARGV[i].class})"
  ```
- The output shows the three lines in `sum.rb` that have the string “puts” somewhere on the line
- A very simple program (given `ll.rb` as a starting point) but it caused a lot of people (including me) a bit of grief

---

gcd.rb

- This project was pretty straightforward
- The solution is to use `puts` statements inside the loop that updates `m`, `n`, and `r`
  
  + if the statements are inside the loop we can see the values computed at each step
- The easy way is to use one call to `puts` for each variable:
  
  ```
  puts m
  puts n
  puts r
  ```
- But this prints three lines on each iteration
- A nicer output is produced if there is just one call to `puts`, e.g.
  
  ```
  puts "#{m} #{n} #{r}"
  ```

This example uses "expression interpolation", introduced in last week’s slides

---

bmi.rb

- This project builds on experience with `f2c` (the fahrenheit to celsius converter)
- The equation for BMI, given height `h` in inches and weight `w` in pounds:
  
  \[
  (w / h^2) \times 703
  \]
- It wasn’t my intention but this project turned out to be a rude introduction to some of the pitfalls of numeric computing....
- The first problem: the answer is always 0!
- Here is what happens, assuming you wrote either of these two expressions for BMI:
  
  ```
  bmi = (w / (h * h)) * 703
  bmi = (w / h**2) * 703
  ```
- Ruby first computes `h^2`
- it then divides `w` by `h^2`
- it then multiplies the quotient by 703

What would happen if you wrote `w / h * h`?
bmi.rb

So why does this expression always seem to be 0?

\[
bmi = \frac{w}{h^2} \times 703
\]

- when Ruby does operations on integers, it truncates ("rounds down") non-integer values
- for common values of \(w\) and \(h\) the quotient is less than 1
- the quotient gets rounded down to 0, and then \(0 \times 703 = 0\)

This is a pretty devious bug for beginners to uncover

- I should have anticipated it and included some more slides on integer arithmetic
- Even though I didn’t explain truncation, you had all the “tools” you needed to understand what was going on
  - if a program is doing something completely weird, add calls to \texttt{puts}
  - you might need to break one complex expression into several smaller ones
  - you can also do experiments with \texttt{irb}

Using the first strategy, change the code in your program from

\[
bmi = \frac{w}{h^2} \times 703
\]

into

- \texttt{puts w}
- \texttt{puts h}
- \texttt{hsquared = h**2}
- \texttt{puts hsquared}
- \texttt{puts w/hsquared}
- \texttt{bmi = (w/hsquared) \times 703}
- \texttt{puts bmi}

When you run the program you’ll discover \(w / hsquared\) is 0 and you’ll know why \(bmi\) is 0

Here is the transcript of a session with \texttt{irb} that does the same thing:

```
>> h = 65
=> 65
>> w = 150
=> 150
>> h * h
=> 4225
>> h ** 2
=> 4225
>> w / h**2
=> 0
```

To further test the hypothesis that integer results are being rounded down:

```
>> 9 / 10
=> 0
```
### bmi.rb

- A second approach: use floating point arithmetic
- Before you change your program, go back to irb and redo all the previous tests using floating point values for \( h \) and \( w \):

  ```ruby
  >> h = 65.0
  => 65.0
  >> w = 150.0
  => 150.0
  >> h ** 2
  => 4225.0
  >> w / h**2
  => 0.0355029585798817
  >> (w / h**2) * 703
  => 24.9585798816568
  
  - Much better! Go back to bmi.rb, change `to_i` to `to_f` in the definition of \( h \) and \( w \) and you’re done

### Arithmetic Conversions

- When Ruby evaluates an expression involving an integer and a float, it converts the integer to a float and then performs the operation

  ```ruby
  >> 3 * 4
  => 12
  >> 3.0 * 4
  => 12.0
  >> 3 + 4.0
  => 12.0
  >> 3.0 + 4.0
  => 12.0
  
  - Note that mixing floats and integers does not change the order of the operations

- Do some more experiments on your own
  - verify the conversion is applied for `+` and other operators
  - verify it also works on comparisons, e.g. \( 24 < 24.5 \)

- What do you think will be the result of these expressions?

  ```ruby
  >> h = 65
  >> w = 150
  >> (w / h.to_f ** 2) * 703
  >> (w.to_f / h**2) / 703
  >> (w / h**2) * 703.0
  >> ((w / h**2).to_f

  Note that method calls have a higher precedence than any arithmetic operator: \( h.to_f ** 2 \) converts \( h \) to a float before doing the exponentiation

### bmi.rb

- There was one other common problem with the BMI program

- Several people who extended the program to print CDC labels wrote code that looked like this:

  ```ruby
  label = case
  when bmi < 18.5 : "underweight"
  when bmi > 18.5 && bmi < 24.9: "normal"
  when bmi > 25.0 && bmi < 29.9: "overweight"
  else "obese" end
  
  - At first glance this seems to cover all the cases
  - But the problem is this code assumes the values of \( bmi \) have only one digit following the decimal point
  
  - What does this program print when \( h = 65.0 \) and \( w = 150.0 \) (\( bmi = 24.9585798816568 \))?
bmi.rb

- One way to fix this problem is to make sure the comparisons cover the full range of values, e.g.

```ruby
when bmi > 18.5 && bmi < 25.0: "normal"
when bmi >= 25.0 && bmi < 30.0: "overweight"
```

- But the idiomatic way to write this is to have each comparison assume the previous ones fail, so it's not necessary to check the lower bound:

```ruby
label = case
  when bmi < 18.5 : "underweight"
  when bmi < 25.0 : "normal"
  when bmi < 30.0 : "overweight"
  else "obese"
end
```

invoice.rb

- This project was pretty straightforward, and most people did a good job

- The main programming decision is how to print the output lines

  - the code to split the input into three fields is
    ```ruby
    item, num, amt = line.split
    ```

  - One way to print the values is to just print the entire input line, which is already formatted:
    ```ruby
    puts line
    ```

  - The output:
    ```ruby
    % invoice.rb books.txt
    latex         1    23.99
    ajax          3    19.95
    ...
    ```

  - A second way to print the values is a puts expression with each value:
    ```ruby
    puts item + " " + num + " " + amt
    ```

    - Since the separator is the same in each case, you could also write this:
      ```ruby
      puts [item,num,amt].join(" ")
      ```

    - The drawback is that fields don’t line up nicely, e.g.
      ```ruby
      % invoice.rb books.txt
      latex 1 23.99
      ajax 3 19.95
      ```

    - It would be nice to have equal width columns

printf

- This is a situation where the printf method is useful

  - printf, like puts, is a kernel method that you can call from anywhere in your program

  - The first argument to printf is a format string

  - The format string has text to print plus a set of “place holders”

  - The remaining arguments are values to insert into the format at the locations specified by the place holders

  - Example: suppose we want to print lines like “2 @ 19.95” or “12 @ 1.99”

  - We need a format string that has place holders for an integer and a float, separated by the letters “@”

    ```ruby
    >> printf "%d %@%f", 2, 19.95
    2 @ 19.950000
    ```

    - The place holders are items that begin with %
    - The letter following the % tells Ruby what to insert

    `%d`: insert an integer here

    `%f`: insert a float here
printf (cont’d)

- Read more about `printf` in the Thomas book or the on-line documentation
- The key ideas:
  - you can specify a field width after the place holder type, e.g. `%6f`
  - for floats, you can specify the number of digits before and after the decimal, e.g. `%6.2f`
  - to insert a string use `%s`
  - to left-justify the string, use a minus sign, e.g. `%10s`
- Note: `puts` always prints a newline character, but `printf` does not
- There is also a `sprintf` method -- it makes a formatted output string, but doesn’t print it

```ruby
>> s = sprintf "%d @ %f", 2, 19.95
=> "2 @ 19.950000"
```

invoice.rb

- A fancy output format for `invoice.rb`:

```ruby
printf \"%-10s %3d @ %6.2f = %6.2f\n\", \{item, num, num * amt\}
```

```ruby
latex 1 @ 23.99 = 23.99
ajax 3 @ 19.95 = 59.85
tiger 1 @ 13.99 = 13.99
rails 2 @ 24.95 = 49.90
```

mpg.rb

- The main difference between `mpg.rb` and the invoice program is that instead of printing a line, we want to keep track of the car with the best mileage
- The basic plan:
  - before the loop define a variable for the best mileage seen so far
  - give it an initial value of 0
  - inside the loop, update the variable whenever the current car has better mileage
  - after the loop the variable will have the best mileage

```ruby
best_mpg = 0.0
while line = gets
  ... best_mpg = ...
end
```

- Since you also want to know the name of the car with the best mileage, save the name in another variable defined before the loop

```ruby
since you also want to know the name of the car with the best mileage
```
If you’re going to do this, use a hash instead of an array to represent each car:
```ruby
a << { :id => name, :mpg => mpg }
```
The call to sort is then:
```ruby
a.sort! { |x,y| x[:mpg] <=> y[:mpg] }
```
To print the best car:
```ruby
printf "best mileage: %s at %.1f mpg\n", a[-1][:id], a[-1][:mpg]
```
If you use a hash, you can refer to attributes of a car by name:
- You don’t have to remember `x[0]` is the name and `x[1]` is the mileage.
- This code is also easier to maintain, since you can add and remove attributes later.

Some people were unsettled by the fact that this program only reads two strings:
- You don’t need a loop to read the sequences:
  ```ruby
  seq1 = gets
  seq2 = gets
  ```
- A loop is OK, and is probably what you would use for a more complex program:
  ```ruby
  seq = Array.new
  while line = gets
    seq << line
  end
  ```
After this refer to the sequences as `seq[0]` and `seq[1]` instead of `seq1` and `seq2`.

I already gave away the code in the last lecture, as an example of a method:
```ruby
def diffs(seq1, seq2)
  ndiffs = 0
  seq1.length.times do |i|
    if seq1[i] != seq2[i]
      ndiffs += 1
    end
  end
  return ndiffs
end
```
Simply comparing the sequences (`seq1 == seq2`) won’t work:
- This expression compares the entire sequence and returns true or false.

The goal for this program is to make a “gene” and then mutate it a specified number of times:
- My suggestion was to use a String object, but an Array is OK, too:
  ```ruby
  gene = Array.new(100, 'A')
  gene = Array.new(100, 'A').to_s
  ```
- If you want to compare the original gene to the mutated gene, make a copy:
  ```ruby
  mutant = gene.clone
  ```
- The main loop chooses two random numbers:
  - The first, between 0 and n-1, is the index of the letter to change.
  - The second, between 0 and 3, is the new base.
- I suggested using `(rand * n).to_i`, but `rand(n)` also works:
  - `rand` without a parameter is a float between 0 and 1.
  - `rand` with a single parameter `n` is an integer between 0 and `n-1`.